Rockwell
Automation
罗克韦尔自动化技术丛书

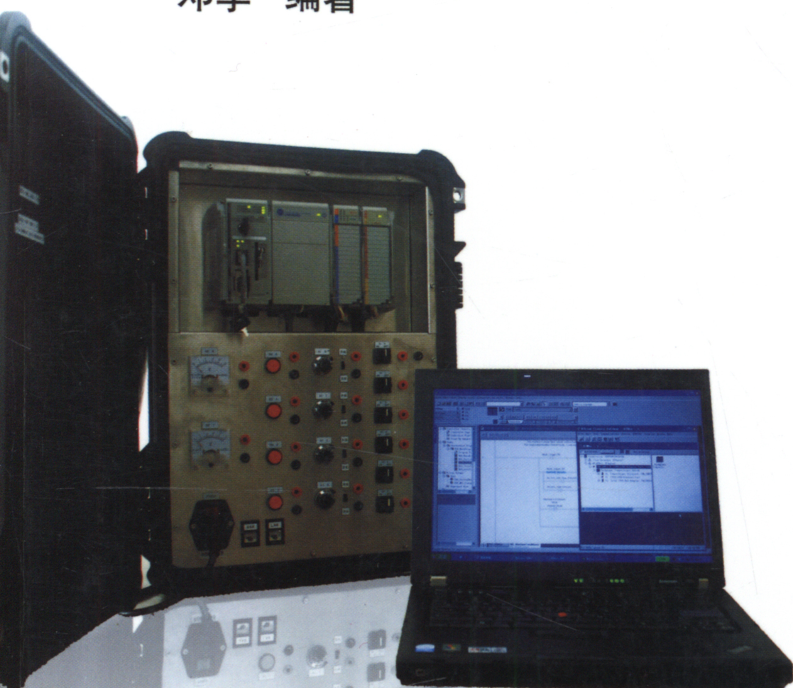


PAC编程基本教程

——适用于罗克韦尔自动化Logix控制器

邓李 编著









机械工业出版社
CHINA MACHINE PRESS



VISIT…

LANZAROTE

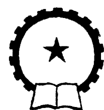
Caliente.COM

罗克韦尔自动化技术丛书

PAC 编程基本教程

——适用于罗克韦尔自动化 Logix 控制器

邓　李　编著



机　械　工　业　出　版　社

本书以罗克韦尔自动化公司的 Logix 控制器为编程平台，结合现场实例的梯级逻辑介绍了指令的运用，由浅入深，循序渐进地介绍了 PAC 的四种编程模式。既有 PLC 的基础指令编程，数组和通信操作的高级指令编程，还有 PAC 的特殊指令和编程模式，内容全面覆盖了当前可编程序控制器的控制功能。本书详细讨论了指令实际运用过程和编程细节是如何推敲的，强调了编程能力和编程习惯的训练，注重标准化编程的发展过程和趋势的研究，是一本极具实用价值的指导性编程教程。

本书适用于学校作为实操能力训练的课程教材，生产企业培训中心针对技术培训的教材，也可作为项目开发编程人员和控制系统维护人员提高自身编程能力的自学教材。

在本书学习进程的编程练习中，建议使用与之紧密配合的标准编程实验设备和编程实验指导书，可以高效地完成各个章节对应的编程训练，保持完整内容的编程训练过程。

图书在版编目（CIP）数据

PAC编程基本教程/邓李编著. —北京：机械工业出版社，2011. 10
（罗克韦尔自动化技术丛书）
ISBN 978-7-111-36030-8

Ⅰ. ①P… Ⅱ. ①邓… Ⅲ. ①可编程序控制器－程序设计 Ⅳ. ①TP332. 3

中国版本图书馆 CIP 数据核字（2011）第 200873 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
策划编辑：林春泉　责任编辑：赵　任　版式设计：霍永明
责任校对：王　欣　封面设计：鞠　杨　责任印制：乔　宇
北京机工印刷厂印刷（三河市南杨庄国丰装订厂装订）
2012 年 1 月第 1 版第 1 次印刷
184mm×260mm · 22. 75 印张 · 560 千字
0 001—3 000 册
标准书号：ISBN 978-7-111-36030-8
定价：85. 00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
社服务中心：（010）88361066
销售一部：（010）68326294
销售二部：（010）88379649
读者购书热线：（010）88379203

网络服务
门户网：http：//www. cmpbook. com
教材网：http：//www. cmpedu. com
封面无防伪标均为盗版

前　言

多年前，我的一个同事和一个邻居相继去了美国，几年后，听说她们做了程序员，干得还不错。说实话，我非常地吃惊和不解，要知道她们在国内学的可是语言，一位学的是英语，另一位学的是日语，典型的文科生。那时，我常常给电气专业出身的学员们作 PLC5 的编程培训，理所当然地认为编程自然应是受过多年理工科训练的工程师们干的活，其逻辑思维的慎密和严谨，专业知识的丰富和熟练，并非一日之功。

直到近些年，第三代控制产品 Logix 控制器的问世，在编程软件人机交互界面的轻松操作中，完全没有注意到编程的简单和容易，为了编写这本《PAC 编程基础教程》，我有意识地研究了计算机控制系统完成执行动作指令的演变和进化过程，竟然改变了原有的看法，尤其是在完成了这本书后，得出的结论是：几乎任何起点的人都可以学习编程！

PAC 控制器，仅仅数点它的功能，便会令人望而生畏，功能这么强大的系统，编程是不是很难？这给我们提出了一个有趣的问题，究竟是复杂的计算机控制系统编程容易？还是简单的计算机控制系统编程容易？事实总是跟我们想像的不一样，只有当我们对原先未知的东西，有了相当的了解之后，才会看到其真实的一面，脱离了之前的武断，恍然大悟。

事实上，指令功能越强，编程所需要的技巧就越少。所谓指令系统的提升，实际上就是将原来需要技巧处理的经验转化为指令的标准功能，一代又一代的产品更新，多年前辈的经验逐步地化作了指令的功能和标准化的编程。理解指令功能也许比构思技巧更为重要，有时你绞尽脑汁想出来的解决方案，原本就有一条专用的指令可直接应用，遗憾知之甚少之余，感叹学习有时比创造性劳动更有意义。

PAC 控制器系统的先进无疑使我们感觉到编程越来越容易，指令功能更强，编程更标准化，资源运用更充足，快速而缩减错误的编程方式不但减轻了工程师们的负担，也使现场调试更为方便和快捷，从而大大地缩短了工期，自然节省了工程成本并提升了生产力，这也是 PAC 产品显著的优势之一。此外，标准化还使项目开发者和系统维护者之间达到了共识，系统维护者解读程序不再是困难重重。

任何系统功能和标准都不是突然产生的，而是经过了长期的演变和积累，功能化和标准化让后人分享了前辈的经验和成果，功能化的处理，标准化的编程，不但让新手更容易上手，更重要的是减少出错，避免了陷阱和隐患，直接运用正确的方式，获得了正确的结果。训练新手的一个重要环节就是学会标准化地处理应用对象。

本书介绍的编程方式和数据处理就有多种形式的标准化模式，Logix 控制器的平台使得标准化更容易实现和更容易表达，特别是用户自定义的结构数据和用户自定义的指令，更是把行业或企业的标准统一起来，将管理嵌入了控制系统，让企业受益匪浅，其作用和影响甚至超过了控制系统本身。

18 年的产品培训经历让我看到一批又一批的学员不断重复相同的错误，却无法帮助他们在短期内解决问题，需要了解的产品知识面越来越广，需要更多的时间花费在组态和更新上，于是编程的基础训练变得越来越少。希望这本书能够帮助读者在自由支配的时间里细细

地阅读和练习，从容地思考和研究，学会程序的编写，会编写程序就会解读程序，只有解读了程序，才有可能理解和精通你所维护的系统。

让我们一起来学习编程吧！

本书参编人员有：谢志雄、陈子平、田毅、向京、余一帆、张建雄、柳小平、刘健坤、梁健、陈尔迎、王占平、王作铭、代勇飞、魏海波、林李智、姜薇、程玉进、张敬山、张宾伟。

与本书同时开发的还有配套的 Logix 平台编程实验设备，以及在编程实验设备上实现编程训练步骤的编程实验指导书。编程实验设备简洁而不简单，涵盖了 PAC 所有的指令功能和编程内容，面板信号和外部信号的灵活组合，具有较强的仿真能力和外联功能。编程实验指导书结合本书中的学习进程，提供了足量的习题练习，这些习题不仅与现场实践运用相关，还具有趣味性，每道习题均有编程提示和测试要点，并提供了极具价值的编程参考，方便了编程者对照学习和拓展思路。由于编程实验指导书从最基本的编程操作入手，具有详尽的操作步骤和完整的操作过程，通过实例学会与编程测试有关的编程软件基本功能运用，即使对 Logix 产品一无所知的学习者也可轻松上手，不仅适合学校和企业培训中心完成计划性的训练，还适合自我训练。

编程实验设备和编程实验指导书，将由厦门罗吉克斯技术咨询有限公司提供，需要者可来函来电联系。电子邮件：logixdemo@yahoo.cn。

作者

2011 年 10 月于厦门

目　　录

前言
第 1 章　PAC 控制器基本性能概述 …… 1
1.1　覆盖广泛领域的控制功能 …… 1
1.2　硬件功能模块化的结构 …… 2
1.3　开放的网络平台 …… 4
1.4　标准化的数据结构 …… 5
1.5　智能化的 I/O 设备管理 …… 6
第 2 章　编程实验设备基本结构 …… 8
2.1　Logix 控制器类型简介 …… 8
2.2　Logix 控制器的内存结构 …… 10
2.3　编程实验设备简介 …… 15
2.4　控制器项目组态 …… 18
第 3 章　梯形图编程基础 …… 27
3.1　梯形图的梯级结构 …… 28
3.2　梯级的预扫描和后扫描 …… 30
3.3　养成标准化编程的习惯 …… 32
第 4 章　位逻辑指令编程 …… 37
4.1　继电器输入/输出指令 …… 37
4.2　单脉冲指令的用法 …… 40
4.3　位的互锁操作 …… 43
第 5 章　计时器指令和计数器指令编程 …… 46
5.1　计时器指令编程 …… 46
5.2　计数器指令编程 …… 59
5.3　计时器和计数器联合使用的实例 …… 67
5.4　采用用户自定义结构数据编程 …… 73
第 6 章　算术逻辑运算指令的编程 …… 77
6.1　一般算术运算指令编程 …… 77
6.2　算术指令编程实例 …… 81
6.3　三角函数运算指令的编程 …… 87
6.4　逻辑运算指令的编程 …… 93
第 7 章　比较指令的编程 …… 97
7.1　等于指令 EQU 和不等于指令 NEQ 的编程 …… 97
7.2　综合比较指令 CMP 的编程 …… 100
7.3　范围比较指令 LIM 的编程 …… 101
第 8 章　传送和转换指令的编程 …… 108
8.1　传送指令 MOV 和屏蔽传送指令 MVM 的编程 …… 108

8.2 复制指令 COP、同步复制指令 CPS 和充填指令 FLL 的编程 …… 112
8.3 位域传送 BTD 指令和字节交换指令 SWPB 的编程 …… 116
8.4 转换指令 TOD/FRD 和 DEG/RAD 的编程 …… 122
8.5 ASCII 码转换指令 STOD/DTOS 和 STOR/RTOS 的编程 …… 126
第 9 章 数组指令的编程 …… 133
9.1 数组算术逻辑运算指令 FAL 的编程 …… 135
9.2 搜索指令 FSC 的编程 …… 139
9.3 数组平均值计算指令 AVE 的编程 …… 140
9.4 数组排序指令 SRT 的编程 …… 142
9.5 查找元素长度指令 SIZE 的编程 …… 143
9.6 数据传送的常规编程 …… 144
第 10 章 寄存器指令的编程 …… 149
10.1 位左移指令 BSL 和位右移指令 BSR 的编程 …… 149
10.2 堆栈指令先入先出 FFL/FFU 和先入后出 LFL/LFU 的编程 …… 154
10.3 顺序器输出指令 SQO、顺序器输入指令 SQI 和顺序器装载指令 SQL 的编程 …… 161
第 11 章 程序控制指令的运用 …… 170
11.1 调用子例程指令 JSR 及 SBR 和 RET …… 170
11.2 事件触发任务调用指令 EVENT …… 172
11.3 跳转指令 JMP 和 LBL …… 173
11.4 循环指令 FOR …… 174
11.5 主控复位指令 MCR …… 179
11.6 禁止中断指令 UID 和 UIE …… 181
11.7 暂停指令 TND …… 181
11.8 恒假指令 AFI …… 183
11.9 空操作指令 NOP …… 184
第 12 章 特殊指令的编程 …… 186
12.1 数组（文件）位比较指令 FBC …… 186
12.2 诊断检测指令 DDT …… 190
12.3 数据转变指令 DTR …… 194
12.4 比例微分积分控制指令 PID …… 196
第 13 章 通信指令 MSG 的编程 …… 205
13.1 分配给 MSG 指令的结构数据 …… 205
13.2 MSG 指令通信的编程 …… 207
13.3 MSG 指令的服务性操作编程 …… 220
第 14 章 系统设置和系统状态指令 SSV/GSV 的运用 …… 231
14.1 访问系统日期时间 …… 232
14.2 访问控制器的任务和程序 …… 234
14.3 访问控制系统中的模块或设备 …… 236
第 15 章 Add-On 指令的创建和编程 …… 245
15.1 Add-On 指令的创建过程和引用 …… 246
15.2 将常规动作转化为 Add-On 指令的实例 …… 257
15.3 将信息处理过程转化为 Add-On 指令的实例 …… 263

15.4　将标准规则转化为 Add-On 指令的实例 …… 270
15.5　将特定数据处理转化为 Add-On 指令的实例 …… 274
第 16 章　PhaseManager 编程介绍 …… 282
16.1　PhaseManager 的创建 …… 282
16.2　PhaseManager 的编程 …… 286
16.3　PhaseManager 的故障管理和调试编程 …… 294
第 17 章　顺序功能流程图编程介绍 …… 297
17.1　建立 SFC 的结构和步的编程 …… 298
17.2　SFC 例程运行的外部操作管理 …… 305
第 18 章　功能块编程简要介绍 …… 309
18.1　累加器功能块组态 …… 309
18.2　简单功能块的综合运用 …… 315
第 19 章　语句结构编程简要介绍 …… 322
19.1　常见赋值语句和结构语句的编程 …… 323
19.2　指令语句编程 …… 326
第 20 章　报警功能的三种编程模式 …… 330
20.1　离散量报警功能的编程 …… 330
20.2　模拟量报警功能的编程 …… 340
参考文献 …… 354

第 1 章

PAC 控制器基本性能概述

可编程自动化控制器（Programmable Automation Controller，PAC）是覆盖工业常规控制的通用型控制产品，是近年来领先工控产品制造业公司推出的新型控制器，旨在取代可编程序控制器（Programmable Logic Controller，PLC）并扩充其功能的代表性产品。产品融合了当今优越的计算机技术和成熟的网络技术，数据无缝连接解决了控制系统之间信息沟通的难题，灵活机动的编程语言和编程模式为多样化编程奠定了基础，与传统的 PLC 相比，在控制功能、硬件结构、通信模式、数据形式和 I/O 管理上都有着本质的区别，作为先进的可编程自动化控制器，应该具备覆盖广泛领域的控制功能、硬件功能模块化的结构、开放的网络平台、标准化的数据结构、智能化的 I/O 设备管理等特征。

1.1　覆盖广泛领域的控制功能

面对一个完整的生产工艺过程所进行的自动化控制，往往是综合功能的控制，不仅仅是顺序时序逻辑控制，还包括了简单的过程控制、精确执行的运动控制和随动伺服驱动。在传统工业控制产品中，要将这些相互分离的控制系统整合在一起，表达生产工艺过程的完整信息结构和传递，必须完成各个控制系统之间的通信连接，这在封闭网络协议和数据形式自立的局势下，是极其困难有时甚至是无奈的，为实现接口和数据交换的所花费的繁杂工作，有时超过了控制系统本身的面对控制对象的工作。为此，各个工业控制产品制造厂家在硬接口和软接口上，耗费了大量的人力物力，仍然是收效甚微，兼容往往限于合作伙伴和常规网络之间，综合同一生产工艺过程的所有控制系统的所有数据交流于一体，几乎是不可能的。

可编程自动化控制器的控制功能涵盖了工控系统几乎所有的控制功能——时序逻辑控制、过程控制、运动控制和伺服驱动控制。

时序逻辑控制沿用了 PLC 的优势，移植了成熟且功能强大的逻辑处理的指令系统，并有不同程度的改进，有更为精密的计时器用于更为精确的控制；将文件处理进化为多维数组处理；增加了 ASCII 码的处理功能；增强了通信指令 MSG 的功能，大量的服务性指令操作，更方便系统的自我管理和监视；增加了专用的信息处理指令 SSV/GSV，访问系统的硬件和软件，获取它们的状态和设置组态数据。

由于控制器系统采用了结构化数据，过程控制中包含大量参数的功能块，可以毫无阻碍地移植到控制器指令系统，完成功能块组态和参数调试。具有 DCS 系统丰富特色的 PID 环控制、累加器、高通滤波和低通滤波、选择器等功能块，弥补了单一功能逻辑控制的不足，用梯形图指令完成这些功能，也许需要编写大量的梯级逻辑才能实现，在此，一个诸多参数

集中于功能明确的控制块中便可完成。

专用的运动控制模块与控制器紧密关联，位于同一机架或集成于一体，令信息传递快速而准确，内建在控制器指令系统中的运动控制指令，将传统运动控制的复杂组态过程演变为简单的梯形图逻辑编程，易于理解和掌握的运动控制指令实现运动控制功能，使更多的工程技术人员不需更长时间就能运用自如，极有效地缩短了开发和调试时间。

驱动控制功能改变了传统产品的控制器与驱动器之间通过通信连接，从驱动器获取状态信息，在控制器中完成逻辑控制过程，再将控制结果送至驱动器的做法。直接在驱动器中内嵌的控制功能，不但可以通过专用的功能块组态驱动器的控制模式，还可编制梯形图时序逻辑，完成时序逻辑控制，极大地改善了驱动性能和提高了本地逻辑处理能力，驱动控制系统的快速性和可靠性得到了提高。

最重要的是，可编程自动化控制器将时序逻辑控制、过程控制、运动控制和伺服控制的功能集成在一起，共同使用一套 I/O 模块，共同使用一套数据库，共同面对外部设备访问，数据的无缝集成，使得满足通用型的控制系统达到了完美的境地。

可编程自动化控制器不仅仅具有控制功能全面覆盖的优势，基于优越的操作系统和多样化的编程模式，控制器可以轻而易举地将标准化编程注入其内。如满足适合批处理控制的 ISA S88.01 设备和配方模块，以及适合机器控制的 PackML 方案，可以用简单组态的方式建立起标准模式，实现标准化编程，这给解读程序和维护系统带来了极大的便利。系统还允许创建用户自定义指令，用途广泛的用户自定义指令不但能满足大量重复使用的实例简单化操作，还可以实现标准化运用推广，对于专用系统开发商，封装自定义指令的内容，以保护自己的知识产权，更是大受欢迎。

相比传统 PLC 单一的程序结构和有限的中断调用，可编程自动化控制器的程序文件具有任务、程序和例程三层结构，是较为复杂和运用灵活的。多任务执行的组态和优先权安排，使定时中断调用和事件中断调用可多次响应并优先有序；程序核心结构完整，内部数据库和例程自成一体，程序相互独立并数据隔离；例程编程语言多样化，可根据控制需求或控制目的选用；软控制器可外联计算机编程语言例程，更是给开发人员提供了广阔的应用。

1.2 硬件功能模块化的结构

这里提到的硬件功能模块化的结构，并非传统产品意义上的硬件模块化结构。传统产品的硬件也是模块化的结构，但是硬件的功能并未模块化，处理器或适配器与 I/O 模块有规则地组合在一个框架中，框架内所有 I/O 模块别无选择地成为同一个 I/O 功能组块，并以框架为单位，通过处理器或适配器建立对外的通信。相对功能而言，硬件 I/O 模块都不是独立的，处理器或适配器通过内部数据线跟每个模块的数据缓冲区交换 I/O 数据，这些模块化的硬件综合而成一个功能模块，每个硬件 I/O 模块没有独立的功能，不能单独对外通信。

可编程自动化控制器的硬件功能模块化结构，是指每一个硬件模块都是功能独立的设备，表面看上去跟传统产品一样的硬件模块组成的框架，每个模块在功能上却是相互独立的，这个系列的所有的模块，不管是控制器模块、通信模块或 I/O 模块，都可作为通信的独立单元，都具有发送信息和接受信息的能力，就像一台设备，无论位于哪个槽位，都不会限

制或影响本身具有的功能。

在可编程自动化控制器的系统中，将控制器称为 CPU 或 CPU 模块的说法，变得非常地不恰当，并非只有控制器才带有 CPU 芯片，而是每个模块都带有 CPU 芯片，都是智能模块，具有独立的数据处理和通信管理能力。位于框架的所有模块，不但通过背板获取电子设备必需的供电，还通过框架背板的控制总线功能与其他模块或设备建立通信连接。所有模块插放的槽位是不受任何限制的，即使是控制器模块也可以插放在任何位置，而不必像传统产品那样，必须插放在框架的最左侧。由于硬件模块的相互独立，在同一个框架插放多个控制器或多个通信模块也是允许的。

只要在控制器中进行的 I/O 组态，明确了通信模块和 I/O 模块的从属关系，建立与它们的通信连接，指定与它们的数据交换关系，或通过背板，或通过网络，都可自由地进行数据交流。这样说来，位于同一框架的模块，或许它们之间有着紧密的通信关系，或许它们毫不相干，它们之所以位于同一框架，完全取决于它们所需驻留的物理位置。

同时，可编程自动化控制器的面板上不再建造复杂的通信口，只保留了一个简单的编程口，留与编程终端直接连接，这正是称为控制器而不称为处理器的原因。所有的通信口都被做成独立的不同类型网络的通信模块，像 I/O 模块一样插放在框架上，通过背板跟控制器建立通信连接，并作为控制器的通信通路，延伸到网络。这样，转换网络类型只需更换通信模块，扩展网络也只需增加通信模块，系统结构变更和拓展的实现变得非常容易。通信口的损坏只需更换通信模块，而不会像传统产品那样兴师动众地要更换处理器。

毫无疑问，这从根本上改变了控制器与 I/O 模块的关系，摒弃了传统产品控制器主从式的轮询 I/O 扫描方式，所有的 I/O 模块以通信的方式与控制器交换 I/O 信息，将传统的被扫描变为主动的发送数据，每个 I/O 模块都可单独定义自己的 I/O 数据更新时间，这使得最为节省通信资源的 I/O 数据交换模式周期刷新（模拟量）和逢变则报（离散量）得以实现。由于采用生产者/用户方式来交换 I/O 数据，还可以实现输入数据共享，多个控制器访问同一输入模块只需简单地组态便可以实现。

这样，I/O 模块作为一个通信连接，和外部设备通信（如人机界面访问）是完全一样的，它们一起共享控制器的通信资源，可以将用于 I/O 模块的通信连接和对外通信的通信连接，按照需求进行分配，灵活充分地运用了控制器资源。对于 PAC 控制器，很少严格地去讨论能带多少个 I/O 点，具有多大的带 I/O 的能力，我们所关注的是控制器的连接限量是如何被运用的，这个通信运用包括了硬软件的共同分享。

不管怎么说，控制器毕竟是一个非常特殊的模块，它除了含有每个模块都具有的 CPU 芯片，有的控制器模块甚至使用了两个 CPU 芯片，还有内存用来存放执行代码和操作数据，它的操作系统使它当之无愧地成为整个系统的核心部件，所有的数据都集中在它的内存空间，接受内存执行代码的处理，并发往其他设备。

的确，在这个系统中，模块和设备的区分不是那么明显，至少在数据交换上来说，它们几乎是等同的。如果非要细究，区别在于硬件模块是框架组合，模块在供电上并非独立，不同于设备的独立取电，但这也正说明硬件的系统集成紧密。总而言之，要强调的是硬件功能模块化，通信功能的独立。

1.3 开放的网络平台

开放的网络是可编程自动化控制器的一大特色。工业控制系统的网络数据交换给人们带来的困扰由来已久，只有采用开放的网络协议，才能让广大供应商的产品共聚在同一个网络，为无缝连接的数据交换提供共同的物理平台。现在，致力于工业控制系统的产品开发的生产商，协同商议促使开发，在开放网络协议方面已达到了共识。可编程自动化控制器系统选择了开放式的网络，也就是选择了数据无缝连接的平台。

针对工业控制系统不同需求的信息交换层次，可编程自动化控制器拥有不同层次的开放网络。一般来说，按目前的信息交换需求分层，有信息网、控制网和设备网。

信息网是适合大量数据采集的网络，主要用作于人机界面对控制器的数据采集，人机界面越来越多和越来越高的需求，促使了信息网在工业控制领域的发展；将控制系统数据嵌入管理系统数据库，关联数据所需的信息交换满足大批量数据处理，也成为目前需求的一种趋势；控制器之间的数据交换，也可以通过信息网传送，通常是一些非预定性数据。信息网通常用 EtherNet/IP 网络实现。

控制网是专为工业控制需求而设计的网络，主要用作预定性数据的传送，能够确保实时数据交换的可靠和及时，控制器与远程 I/O 模块数据的交换。控制器和控制器之间连锁信息的交换，都是控制网数据交换的主要需求。由于网络速度快和网络规划的确保，具有极好的响应速度和可靠性；由于网络组态和网络管理被设计得完善和周全，控制网的网络优化极易实现。控制网通常用 ControlNet 网络实现。

设备网是直接面对现场传感设备采集数据的网络，主要用作现场设备数据采集，将传统传感器产品的模拟信号，改为网络通信的方式直接获取数据，它的出现不但挑战了传统的 I/O 模块的数据交换模式，同时也刺激了智能化传感器的发展。设备网络是一个特殊的网络，不但要有数据通信的网络连线，有时还要为现场设备供电，是将通信和供电集于一体的网络，所以网络电缆和各种接头的选型较为复杂。设备网通常用 DeviceNet 网络实现。

这些开放式网络都是智能网络，能够规划和管理网络数据的交换，通过专用的组态软件对网络设备进行组态，不但能让网络数据流通获得最佳性能，还可以实现网络的数据流量的监视和故障的诊断。智能网络是需要组态的，所以这些网络都有相应的网络组态软件，网络组态软件自然成了可编程自动化控制器系统软件组合不可分割的一部分。

网络的互通性能也是考查开放性的一个重要指标，被称为控制总线（ControlBus）结构的框架背板充当了网络网关的角色，插放在框架中的相同类型或不同类型的通信模块，通过背板从一个网络转移到另一个网络，不需要作任何组态或设置，即使是不同类型的网络也没有任何障碍，如此的框架背板几乎可以称为万能网关。这样，本地网站点和远程网站点之间的数据交换变得自如流畅，不再像传统网络那样极为困难和繁琐，这要感谢硬件功能模块化结构和背板控制总线结构带来的好处。

由于良好的网络互通性能，这三层按应用分类的网络在运用实际的网络时，并不十分严格和界限分明，往往是相互交叉和相互延伸的，有时同一网络同时兼任了信息层和控制层的需求；有时同一网络同时兼任了控制层和设备层的需求；有时甚至同一网络同时兼任了 3 个应用层的需求。根据实际应用所进行的网络选型，一个应用可能会产生多种方案，这就是开

放式系统带来的灵活性。

目前，有的产品甚至开始尝试一网到底的做法，让一种网络同时兼任所有的信息交换工作。被看好的无疑是工业以太网，它不但拥有以太网容量大、速度快的优良品质，还可借用广泛的媒介质设备节省硬件开发的时间和成本，和其他领域通用的便利更是受到人们的欢迎。它面临的问题只有一个，就是将工控系统中预定性数据，即 I/O 数据如何优先传送。这个问题似乎得到了解决：网络的中转设备选用智能的交换机。作为网络数据集中交换设备，智能的交换机可以根据数据类型作出选择，让优先级别高的数据从优先通道先行，此外，它还有强大的诊断能力对网络数据流通进行监视。

1.4　标准化的数据结构

实现信息交换无缝连接的另一个重要条件就是采用标准的数据结构，仅仅有开放协议的网络只保证了信息的无碍沟通，但不能保证通信设备之间的数据识别，只有彼此采用相同的数据结构，才能确保数据的有效识别和直接使用。工业控制系统无需特别地去制定一套通用的数据结构标准，工业控制系统的计算机化，以及人机界面数据库和嵌入的管理系统数据库的日益增多的需求，别无选择的是计算机标准数据结构。

可编程自动化控制器的数据形式采用的是计算机的标准数据结构。习惯了的逻辑控制简洁直达的地址形式转为计算机标签的数据形式，还需要一段时间去适应，以至于常常称标签数据为地址，甚至感到建立标签的繁琐，但最终能接受并感受到标签数据的优越之处，特别是在系统架构非常庞大，数据交换十分复杂的时候尤为如此。

在控制器数据库中被称为标签的数据名称，可以直接用 I/O 设备名称，或控制中间变量直接命名，字母文字简要地表达了具体的控制对象或控制过程含义，故又被称为注释性标签。例如外部电机设备可以直接表达为标签 Motor，中间控制变量运行位可以直接表达为标签 Run，传统外置的纸质 I/O 注释文本说明被撇弃，设备名称终于跟控制器数据融为一体。随着计算机操作平台多元文字平台的功能增强，基于微软平台的编程软件中，中文的注释说明附属在标签之后，给阅读程序带来了极大的便利。

标签的数据类型可以是基本数据，计算机基本数据的两大类离散量和数据量，用作编程对象的操作数。离散量的表达形式正是逻辑控制应用最多的布尔量操作数，现场开关设备和逻辑关系的内部位都用到了布尔量；数据量的形式较为多样化，以字节为单位有短整数、整数、双整数和实数，可根据不同的应用需求选用，一般推荐双整数和实数，以缩减控制器资源，作为 32 位基本数据单元的控制器，指令中使用系统的基本数据单元，无论是空间还是时间都是最为节省的。

标签的数据类型也可以是结构数据。结构数据集合基本数据的操作数于一体，作为结构数据子元素的操作数，仍可被访问和检索。结构数据在应用中更为常见，系统预定义的结构数据包括了 I/O 结构数据、专用控制块结构数据、功能块结构数据及运动控制结构数据。I/O 结构数据标签是组态 I/O 模块时创建在控制器数据区域的，每个模块都有不同的数据结构，包含了模块的组态信息、状态信息和 I/O 数据；专用控制块结构数据是指令特定的数据结构，不同的指令有不同的数据结构，比如计时器指令有计时器结构数据与之对应，计数器指令有计数器指令与之对应，较为复杂的 PID 指令和 MSG 指令的结构数据就更为复杂了；

功能块数据结构来自于 DCS 系统的仪表参数，每个功能块都对应一套十分复杂的结构数据，这反而令功能块功能和参数意义十分明确，组态特别方便。运动控制结构数据自然和每一条运动控制指令息息相关，每一条运动控制指令都要分配一个结构数据标签，用来记录运动控制的运行情况。

最为丰富的尚推用户自定义结构数据。用户自定义结构数据是为了满足各种应用需求，用户自己创建的结构数据，在项目应用中十分有价值，一个包含各种数据类型的结构数据可将相关信息紧密地集中在一起。有时是围绕一个控制对象而创建，将不同类型的操作数合并在一起，方便编程索引和数据查找监视。有时为了数据的传送和变更，可以将近似刷新频率的数据集合在一起，共同更新或传递数据，避免了数据无效交换的损耗。有时为了人机界面的访问，将被访问信息集中在一个结构数据，仅需为通信提供一个连接，极容易地就实现了数据捆绑的优化组合，而不需要额外的处理，这对需求日益增长的人机界面访问来说，提供了一个高效率的通信平台。大多数用户正是在用户自定义结构数据的使用中尝到了标签数据的甜头，从而由不习惯转为乐此不疲。

近年来推行的集成架构所提倡的一套数据库，即下层设备、控制器数据库和人机界面数据库共享一套，这一套数据库就是控制器中的数据库，结构化数据不但提供了一个标准在可编程自动化控制器系统内部及其他设备之间交换，还可以让人机界面通过指向功能直接访问，而不需要在人机界面中再建一套数据库，更重要的是还为二级计算机管理系统直接提供了数据库。可以看出，在控制器中的结构化数据表，正是计算机系统中的关系数据库，一个结构数据标签对应了关系数据库的一条记录。

无论是基本数据还是结构数据，都可以建立一维至三维的数组，除了控制器中拥有数组处理的指令可在内部进行处理，数组还为上位计算机直接提供了符合计算机处理规则的原始的多维数据。

通过结构数据标签的沟通，工业控制系统与常规计算机在数据上已经融为一体，这真是个皆大欢喜的结局。

1.5 智能化的 I/O 设备管理

能够在编程软件上直接监视控制器属下的所有 I/O 模块，乃至与之通信的所有设备，恐怕是现场维护人员最为企盼的事了，可编程自动化控制器系统轻而易举就实现了这个需求，这得益于智能化的 I/O 模块和通信模块，可以响应通信并进行自我诊断的模块，为外部请求提供自身的状态信息并非难事。

与传统产品不同的是，编程软件的 I/O 组态项是非常重要的一部分，这里组态的 I/O 模块，不但建立了控制器和所属模块的数据交换的连接，还提供了监视模块的页面，每当模块与控制的通信连接发生了问题，报警的符号赫然出现在这个模块上，点击进入便可读到相关的信息，并提供相应的故障代码。

每个模块专用的模块信息页面，可以读到模块的名称、型号、版本、系列号、生产厂家，还有模块的当前数据传送模式、主要故障和次要故障代码、拥有者状态、模块组态状态、电子识别状态以及时间硬件和同步状态。

每个模块还有一个背板状态页面，显示模块通过背板传送数据的状态，发送或接受的错

误计数器，为诊断提供了直接的信息。

每个模块的左下角，总是显示模块与控制器的数据交换状态，几乎一眼便知模块是否正在跟控制器交换数据。

在编程软件中组态 I/O 模块时，将在控制器数据库产生与之相匹配的 I/O 结构数据，有别于传统产品的 I/O 模块数据交换，结构数据中不但含有被用于控制程序的 I/O 数据，还含有送至模块的组态信息和从模块读回的状态信息，不管输入模块还是输出模块，模块和控制器之间交换的数据都是双向的，通过模块与控制器的连接，频繁交换如结构数据中的数据内容的一个数据块。

维护人员可以通过编程软件在线监视模块状态，直观地读取模块的工作状态，由于 I/O 模块结构数据的数据提供，也可将数据结构中的状态信息数据从人机界面访问获取，编辑成操作员便于理解的说明，于操作员界面显示，让操作员直接了解系统状况，甚至可作即时处理。操作员进行的处理操作，可由梯形图的梯级逻辑实现，MSG 指令的服务性指令操作可提供各类命令执行，无异于编程人员在编程软件上的操作。

智能化的 I/O 模块带来的另一个惊喜是，离散量模块带有开路诊断功能和输出电子保护电路，可以对每一个通道分别进行管理。离散量输入模块通道的开路锁定，对于接触不稳定的输入回路是最好的判断手段；离散量输出模块通道开路的脉冲测试，亦可获得模块信号回路开路的状态。离散量输出模块的电子保护回路将在输出回路过电流时迅速关闭输出回路，从而保护模块不被烧坏。

最具实用意义的是对输出模块的输出回路在非正常状态时所作的处理。在控制器非正常工作状态或与 I/O 模块的通信失去连接时，也即控制器对输出回路失去控制时，可以设定此时每个通道的输出状态。对于离散量输出模块来说，可以设定输出回路处于接通、关闭或保持状态；对于模拟量输出模块来说，可以设定某个特定的输出量或保持在原值。

值得一提的是，可编程自动化控制器系统的任何一个模块，包括 I/O 模块在内，没有任何跳线和组合开关需要设置，这些传统产品用来组态的硬件，统统被软件组态所代替，这是因为所有的模块都是智能的，能接受组态信息并给予相应的处理。当然，这将引起关注，这些模块是如何保持它们的组态信息的。一般来说，I/O 模块都是通过与控制器建立连接时，临时获取组态信息，如下载项目到控制器、模块所在框架上电、在线修改组态的确认和 MSG 指令执行组态，这些组态信息一直在线保留在 I/O 模块中，当框架关闭电源时，所有的组态信息将全部丢失，等下次与控制器连接时重新获取；通信模块与控制器的所属和连接关系在与控制器建立连接时获得，与 I/O 模块相似，但是通过网络组态软件或连接软件所获得的组态信息，会闪存在通信模块之中，即使离开供电的框架，也不至于丢失组态信息。

以上对 PAC 控制器的基本性能进行了简要的介绍，旨在我们学习编程之前，对程序运行的硬件环境有一个初步的了解，从而知道，根据需求变化和生产技术的进化，传统产品发展到当前的新型产品所传承和拓展的产品功能，这就是为什么我们既要学习传统产品的逻辑控制编程，又要学习新型产品的特殊编程的原故。

第 2 章

编程实验设备基本结构

罗克韦尔自动化有限公司开发、应用和推广 PAC 产品已有 10 多年之久，系列产品已趋于成熟，拥有从上到下完全的系统产品集成架构，无论是硬件结构还是软件配备都十分丰富和完整，在工业控制行业内极具代表性和占有领先地位。

作为罗克韦尔自动化集成架构的核心部件，Logix 控制器具有所有的 PAC 的性能特征。其先进的控制功能和优良的网络性能以及编程软件良好的交互界面，使得功能强大的系统具有一张平易近人的面孔，如此简洁明了和通俗易懂的学习过程，不但令初入门的新手受益匪浅，也深得项目开发工程人员和现场维护技术人员的认同和喜爱，近年来逐渐在各个行业和领域得到广泛的应用和推广。

本书的编程练习和实例讲解，全部基于 Logix 控制器系统平台和常规网络进行，使用 Logix 控制器的编程软件和相应的连接软件。我们将运用编程终端，经历实际的软件操作和针对应用需求的实例编程，通过控制器和 I/O 模块及编程设备外接的开关量和模拟量信号，模拟真实的运行场景，用来检查和调试编写的程序执行代码，不但能得到循序渐进的系统的编程训练，了解和熟悉系统的编程指令，还将大幅提高现场操作的实际技能。

2.1 Logix 控制器类型简介

为了适应各种需求、价位和场合，像所有的系列化产品一样，Logix 控制器也有分类，应该针对现场控制对象和控制过程，根据不同的应用目的和应用层面分析，以及项目的投资情况，选定适合的产品或系统结构。

Logix 控制器系列有 5 种控制器类别可供选型：

- CompactLogix 紧凑型控制器
- ControlLogix 完全型控制器
- DriveLogix 驱动型控制器
- FlexLogix 分布型控制器
- SoftLogix 软件型控制器

如图 2-1 所示，是 5 种控制器的集合。

CompactLogix 控制器是紧凑性的控制产品，硬件结构精巧而紧密，系统功能完整而丰富，较 ControlLogix 控制器价格更为经济，拟用来代替 SLC500 控制器系列。它的发展非常像当年的 SLC500 系列的控制器，这是一个起点较低，最后发展到接近高性能产品的典型例子，就像 SLC500 控制器系列最后发展到接近 PLC5 的功能一样。CompactLogix 控制器的发

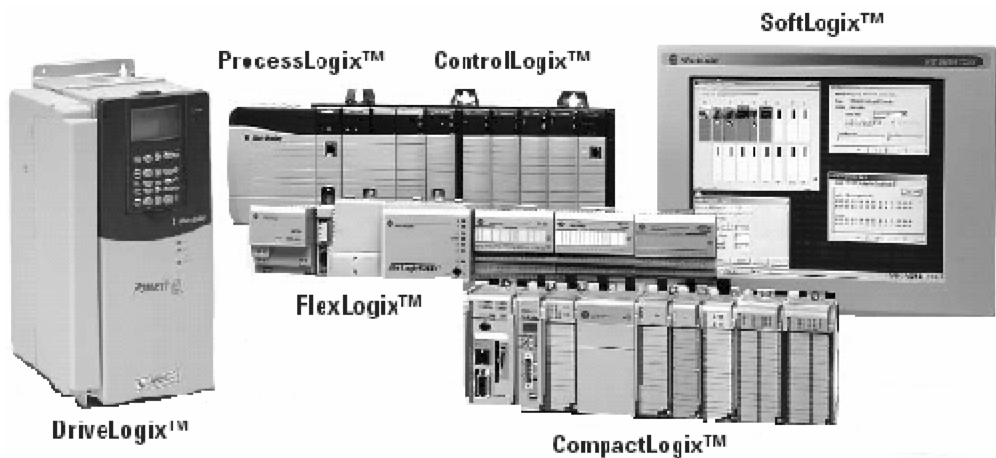


图 2-1　5 种控制器的集合

展，今天也到了接近 ControlLogix 控制器的功能，最为明显的表现是 1768-4X 系列产品的问世，这个产品开发了运动控制功能，这使得 CompactLogix 与 ControlLogix 相比较，除了规模上的差异之外，在功能上几乎没有差别。一网到底的概念最早在 CompactLogix 控制器系统上推行实施，并运用日益广泛，这使得在网络运用上较 ControlLogix 更为方便和经济。尤其是编程的指令系统，跟 ControlLogix 控制器毫无差异，这表现出控制功能上的一致。CompactLogix 的 I/O 模块最初沿用了 1769 系列的 I/O 模块，即 MicroLogix 1500 的扩展 I/O 模块，这不免带有一些传统产品的特征，直到近年来伴随 1768-4X 控制器推出的 1768 系列的 I/O 模块，才具有一些令人欣赏的 PAC 新特性。

ControlLogix 控制器是 Logix 控制器系列中 PAC 功能最齐全的产品，第 1 章中所谈到的 PAC 的全部性能特征，几乎是基于 ControlLogix 控制器的讨论。可以说 ControlLogix 控制器具有最完整的功能和最全面的特性，它的硬件组合最方便灵活，极具柔性且功能强大的，通信组合和扩展最为便利和丰富，具有最强的兼容能力，作为用来替代 PLC5 系列的产品，性价比却远远超越 PLC5 系统。至今，无论是较大的新项目开发或是 PLC5 项目升级改造，ControlLogix 都会作为首选。最重要的是，ControlLogix 控制器拥有一批功能强大的 I/O 模块，这些模块全部都是智能模块，具有独立的对外通信能力和模块自身的诊断能力，能够跟外部设备，特别是控制器交换大量的信息，不但包括 I/O 数据，还包括模块的组态信息和状态信息。基于系统的网络式结构，控制器与智能的 I/O 模块和通信模块构成了全智能设备的硬件系统架构，表现了优越的柔性和全面的集成性。

DriveLogix 控制器专用于驱动控制，是内嵌在驱动器中的控制器，特定的驱动控制功能块的组态可选择驱动器的控制模式，增强了驱动器的控制功能，本地完成逻辑控制使驱动控制的可靠性和快速性有大幅度的提高。DriveLogix 控制器对外通信跟所有的 Logix 控制器一样方便和灵活，对于变频控制在系统中举足轻重，并需要适当的本地逻辑做快速控制处理。

FlexLogix 控制器属于小型的控制器，用于独立的小范围控制系统。这个基于成熟的 1794 系列的分布式 I/O 模块发展起来的小系统，特别适合不超过 16 个 I/O 模块的分布式应用，由于用 FlexLogix 控制器替代先前的 Flex I/O 适配器，它的硬件结构带有明显的传统产品特征，如 I/O 并无独立对外通信能力，PAC 系统智能 I/O 的某些管理在这里难以实现，但

这并不妨碍它成为物美价廉的小系统的最佳选择。尤其是小巧尺寸硬件结构的优势，更是生产流水线上狭小空间安装的理想运用。

SoftLogix 控制器是运行在计算机微软平台的软控制器，由于软件公用平台可能感染病毒导致停机的运行风险，建议不宜采用，但在某些特定的场合，它却显示出特有的长处，比如某个系统远程 I/O 的状态与计算机内某系统主导软件的运行息息相关，一旦计算机停机，外部 I/O 的执行动作再无意义，并且这个系统控制器与计算机主导软件之间有大量的数据需要交换，选择 SoftLogix 控制器便有了意义，毕竟计算机内部的数据交换无论如何也比最快的网络要方便。此外，SoftLogix 控制器是惟一能够外联第三方例程运行的控制器。

2.2　Logix 控制器的内存结构

所有的基于计算机平台的工控自动化产品都具有内存结构，通常意义来说，内存用来存放程序文件和数据文件。程序文件包含了动作执行代码的组合形式以及执行代码组合的执行顺序和执行模式，甚至包含了与程序处理关联的 I/O 数据访问模式；数据文件是用来存放程序处理之前的数据和程序处理之后的数据，数据的存放方式和数据的表达形式都是程序执行代码访问必须关注的，也即编程指令中的访问地址的书写规则。Logix 控制器的程序文件和数据文件有独到之处，不能沿用早期传统产品的经验，在学习编程之前，必须对控制器的内存结构了解清楚，因为每一个执行动作都会关联到程序文件和数据文件。

Logix 控制器内存的程序文件是三层树形结构，分为任务（Task）、程序（Program）和例程（Routine）。每个层面都具有特定的作用和意义，了解程序文件的结构，以便更好地规划执行代码，获得合理的运行结构和执行进程。

任务是最上层的结构，定义了执行方式并安排了执行顺序，奠定了控制器控制过程的基础。任务有如下三种类型：

- 连续型任务　指的是周而复始地连续执行任务，一个项目只允许定义 1 个，亦可以不定义。连续任务是优先权级别最低的，可以被任何周期型任务和事件触发型任务中断。控制器项目创建时自动产生，可以更改为其他类型的任务。连续型任务在所有的时间都运行，跟系统高层管理的时间片段紧密相关。
- 周期型任务　指的是定时中断执行的任务，周期性的执行任务，须定义周期时间，任务组态要设定优先级别。在规划周期型任务时，其调用的周期时间要小于任务的执行时间，在多个周期型任务执行时要兼顾彼此的时间安排，否则将发生任务调用的交迭。周期型任务的运用常用于需要精确地控制执行时间的场合，或用于长时间才调用一次的执行动作，以节约控制器 CPU 资源。
- 事件触发型任务　指的是事件触发引起的任务调用，事件触发可以是外部输入点变化引起，也可以由 Consumed 标签引起，或事件指令调用引起，还可以由运动控制状态引起，任务组态要设定优先权级别。多个事件触发型任务要兼顾不能发生任务调用的交迭。事件触发型任务的运用可以改善性能和减低消耗，减少控制器 CPU 的资源占用，加快信息的吞吐和系统反映能力，有时也用来捕获正常 RPI 刷新可能丢失的短脉冲信号。

任务的执行顺序是由任务的类型所决定的，连续任务是接连不断首尾相接地执行，只有

周期型任务和事件触发型任务才会中断执行，中断执行遵循高优先权任务中断低优先权任务的原则，一旦中断任务执行完毕，将回到原来的中断点继续执行。在一个控制器项目中定义了多个任务执行，一定要安排规划好它们的执行，确保没有任务执行的交迭情况。

每个任务执行完毕，会将执行的控制结果进行输出处理。自动输出处理在任务属性组态中选定，连续任务和周期任务的默认组态不取消，事件触发任务的默认组态取消。输出数据处理将本次任务执行的控制结果送出，然后通过输出模块的数据交换，送至被控制的输出设备。

任务的执行要留有足够的时间来响应控制器外部非预定性数据访问，因为外部非预定性数据访问是任务执行的余留时间来进行的，否则将引起非预定性数据访问的堵塞和滞留。

程序是任务下层的一个完整的组织结构，具有相对的独立性，能够启动执行代码的扫描和调动例程执行，它包含了：

- 每个程序由一个数据库和多个例程组成。
- 每个程序都拥有一个独立的数据库，在这个数据库建立的数据标签只能被本程序内的例程引用，且都是内部数据。
- 每个程序中必须指定一个例程为主控例程，作为本程序运行的启动例程。
- 每个程序中还可以指定一个故障处理例程，以解决本程序内任何例程运行时引起的故障。
- 其余的例程均由主控例程中编写的调子例程指令 JSR 来调用。
- 未预定程序（Unscheduled Programs）中存放备用或暂不运行的程序，这些程序会下载到控制器中，但不会执行。
- 同一个任务下的多个程序，将按任务下排列的顺序执行，这个顺序可以通过任务的组态页面重新调整。

程序是一个完整的结构，就像传统产品的一个处理器，如果将一个 PLC5 的项目文件转换为 Logix 控制器的项目文件，将对应着一个连续任务下的一个程序。这正是一个新项目创建时的默认情形。

程序只是任务中的一部分，一个程序全部扫描完毕是不会进行输出处理的，将执行代码划分为不同的程序完全出于非结构的需求。程序划分的主要作用有时用以区分执行不同的控制区域和不同的控制对象；由于不同的程序可以拥有独立的数据库，有时程序起到了隔离数据的作用；在多人合作的大项目中，程序甚至用来区分不同的编程者。

多个程序共同引用的公共数据，应该存放在控制器区域的数据库中。

在任务下，并列着等同于程序的是 Phase Manager，一个完整的设备工作进程管理程序，特定状态模块规定了执行的进程，是面对设备控制过程而设计的。适用于批处理设备阶段性的管理，也是标准编程的一种模式。

位于程序文件结构最底层的是例程，例程是编程体，所有的控制的执行代码都被编写在例程中，有 4 种编程代码形式：

- 梯形图（LD）　最常采用的编程模式，是由梯级组成的程序结构，完成时序逻辑控制的执行代码编辑。通常完成现场复杂的逻辑关系，解决现场机器的连锁关系，明确运动控制指令的执行顺序。指令系统的丰富和强大，还能完成对外通信的信息处理工作，以及维护人员的检查工作。

- 顺序功能流程图（SFC） 多操作的高水平管理，需要将工作进程细化到步，以便于严格控制执行步骤，用 SFC 来编辑步的执行顺序和结构。每个步都是执行代码的组成，激活步的反复操作和非激活步的静止，节约了大量的无效扫描时间。运动控制顺序的准确安排和递进步骤也是人们选择 SFC 的原因之一，通常会结合语句编程运用。
- 语句编程（ST） 采取 ASCII 代码编写程序，通常用于 SFC 的转换条件判断或赋值。可以完成复杂或特殊要求的运算，也可以完成其他编程方式没法处理的数组和表格的数据处理。在语句编程中，同样可实现梯形图指令或功能块的互换执行代码，甚至完成等同的通信操作。
- 功能块（FBD） 组态过程控制，是引入的 DCS 系统的仪表控制组态方式，由功能块之间的连接建立程序结构，参数连接和设定是功能块组态的主要内容。易于完成闭环控制和流量计算等专用功能。在驱动控制中的专用功能块满足各种驱动组态需求，方便而直观。

例程才是控制器程序文件最基本的结构元素，所有的执行代码都是通过例程的不同编程方式编辑而成。例程才是程序文件真正的执行体，只有通过例程的调用和执行代码的扫描，控制过程才得以实现。

根据不同的需求和编程原则，选择不同的例程编程代码形式，并没有严明的界限，也没有严格的规定，甚至没有特别的优劣之分。PAC 的适应性表现在给人们更多的选择和决定权，其中也包括了编程的语言和模式。

Logix 控制器的数据文件是特别不同于传统的 PLC 处理器数据地址的，这直接影响了人们的编程习惯。首先，控制器所使用的数据不再是源自硬件的寄存器地址或字符式的寄存器地址，而是标准的计算机数据标签，这种被称为描述型的标签，由字母、数字和下划线组成，常常具有文字所能表达的含义。这使得 I/O 地址可以映像为设备名称，直接与现场设备挂钩，而无须特别的文档说明，将人们从强记外部设备地址的困境中解放出来。

最初接触标签地址的人们当然还是会有抱怨，因为他们在编程时不能像传统产品那样随心所欲地使用已存在的地址，而需要将一个一个标签在数据库中创建之后才能使用，这似乎耽误了一些时间和感到繁琐，直到他们晚些时候了解可以免去 I/O 文字说明的工作和无须在别处重复建立数据地址，才恍然大悟一次性创建原来是件一劳永逸的事情，从此欣然接受。

Logix 控制器的数据文件并非集中于一处，而是分布在不同的区域，分为全局数据区域和程序数据区域。这两个数据区域的数据引用范围是不一样的，由于区域的划分，可以使得按生产过程和程序功能分类的数据在查询和引用上更为清楚和方便。

数据区域划分原则：

- 全局数据区域 又称控制器数据区域，它含有全部的对外数据和公用的内部数据，其数据可被控制器内所有的例程引用。所有的 I/O 数据和外部设备访问数据都只能或被要求建立在控制器数据区域；规划时要注意只有公用的内部数据才放在控制器数据区域，不要滥用宝贵的全局数据资源。
- 程序数据区域 全部为内部数据，其数据只能被本程序内的例程引用。各程序之间的数据区域是隔离的，以防止标签命名的冲突。程序中的例程所使用的数据，应该集中地放在程序数据库中。特别是指令中使用的结构数据标签，不要随意地创建在

控制器区域，占用了公共资源。

在一个控制器项目中，全局数据区域的数据库只有一个，程序数据区域的数据库却有多个，每个程序都会分配一个数据库，因此有多少个程序就有多少个程序区域的数据库，每当在指令参数中分派数据标签时，总会出现一个全局区域的数据库和一个程序区域的数据库提供挑选。

无论是在全局区域的数据标签还是在程序区域的数据标签，在创建的时候，都要选择数据类型。Logix 控制器的数据的类型细分可以说太多，但粗分只有两大类型，基本数据类型和结构数据类型。

基本数据类型具有如下几种：

- BOOL　布尔数　占用一位　数据范围：0 或 1
- SINT　短整数　占用一个字节，数据范围：－128 ~ +127
- INT　整数　占用两个字节，数据范围：－32768 ~ +32767
- DINT　双整数　占用 4 个字节，数据范围：－2147483648 ~ +2147483647
- REAL　实数　占用 4 个字节 数据范围：$3.4\times10^{-38}\sim1.17\times10^{+38}$（负数）

  $1.17\times10^{-38}\sim3.4\times10^{+38}$（正数）

基本数据也称为操作数，是具体的操作对象，除了布尔量，其余都是数据量，只是使用字节不同，具有数据范围的差异。数据范围小，耗用字节少的不一定节省内存。Logix 控制器的 CPU 芯片是 32 位的，这意味着 32 位是 CPU 处理的基本数据单元。

每个基本数据类型的标签都要占用一个数据单元，哪怕是布尔量，其内存耗用如图 2-2 所示，灰色部分为内存耗用，空白部分为闲置。当数据类型为 BOOL、SINT 和 INT 被分配给一个标签时，控制器也要花费一个完整的数据基本单元，剩余部分则被闲置，可以看出布尔量是浪费得最厉害的，双整数和实数是最节省的。

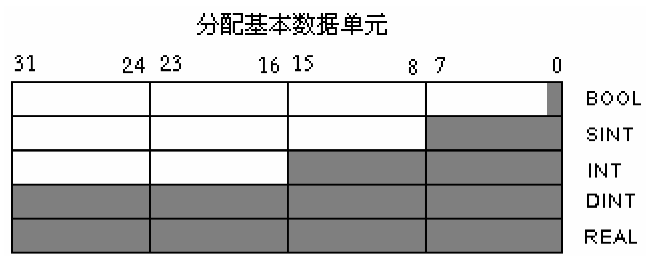


图 2-2　内存耗用

只有使用基本数据单元处理数据和通信才是最节约内存和操作时间的。以后编程使用的指令参数，但凡遇到字的处理，要尽可能地使用双整数和实数，这也是优化的原则之一。

结构数据则比较复杂，分为系统预定义的结构数据和用户自定义的结构数据，系统预定义结构数据是固定不变的，用户自定义结构则千变万化，不同的用途不同的对象将建立不同的结构数据。

系统预定义的结构数据是系统固有的结构数据，往往产生于系统设计之初，为了不同的应用目的，通过开发、演变或移植而形成的。

系统预定义结构数据可分为如下几种：

- I/O 模块组态时产生的 I/O 数据　每个与控制器交换数据的 I/O 模块，都有固定的

内容，诸如组态、状态输入、输出的双向数据传送，不同的模块含有不同的内容，这在模块开发之际便被设计。当在 I/O 组态的树形结构下创建一个模块时，控制器数据区域便会产生一个相应的 I/O 结构数据，该数据结构与模块交换的内容相符。

- 出自于 PLC5/SLC500 的多字元素文件　多字元素其实就是结构数据的初期形式，为了把指令操作相关的数据内容集合在一起，使用了多个字组合。例如被称为三字元素的计时器和计数器文件，在 Logix 控制器中被称为计时器结构数据和计数器结构数据。至于较大的数据块，如 PD 和 MG 的多达十几个字或几十个字的多字元素，也转为结构复杂数据繁多的 PID 结构数据和 MESSAGE 结构数据。
- 运动控制的数据结构　正是结构数据的出现，才使得运动控制得以用指令的形式进行操作，因为一个运动控制的动作，需要相当多的数据来对应，能把这些数据集合在一起的，非结构数据莫属了。复杂的运动控制，其结构数据中甚至可以内嵌上一个数组，用以模拟一条凸轮曲线。
- 过程控制功能块的数据结构　来自功能块的结构数据，可以说是由来已久，DCS 系统历来就是结构数据，被称为一个 Point 的数据，其实就是一个货真价实的结构数据，功能块操作被轻而易举地引入 PAC 就不足为怪了，因为 Logix 控制器的结构数据完全满足了功能块数据形式的需求，并且有更为清晰的表达形式。
- 系统组态信息和状态信息　Logix 控制器没有系统给出状态数据文件，因为系统中不仅仅是控制器，所有的与控制器通信的模块都含有大量的组态信息或状态信息，用户可访问的不仅仅是控制器内部的信息，而是系统的各个模块或设备的信息。不同的系统有不同的组成结构，含有不同的模块或设备，用户可以各取所需地访问信息，Logix 控制器特定的 SSV/GSV 指令就是用于这个目的。面对系统要设置的数据或要获取的数据，都是通过结构数据来存放的，这需要建立相应的结构数据标签。

用户自定义结构数据是由编程者根据各种需求建立的数据类型，一旦定义完成便被嵌入系统，在数据库中创建标签时，在数据类型选项中便可以选择新定义的结构数据类型。

用户自定义结构数据的创建常常出于如下的考虑：

- 针对控制对象而创建　将某个控制对象相关的参数集合在一起，即使是不同的数据类型。作为子元素的各个参数可以是基本数据，也可以是结构数据，还可以是已经定义的用户自定义结构数据。这是一种最常见的用户自定义结构数据，由于其集中和综合性，编程或监视的检索极为方便。
- 针对通信对象而创建　控制器的对外通信，是以标签名作为连接建立通信的，一个标签，即使含有大量的参数数据，对通信而言，也只是占用一个连接，当外部设备跟控制器建立起连接，便可一次交换更多的数据，对比单个操作数的轮询访问，大大地节约了通信资源和节省了网络的带宽，是目前大力推荐的先进的通信模式。
- 针对人机界面访问而创建　人机界面访问的数据，在控制器中被抽取并集中在动态缓存区，以便快捷地访问，如果将同时访问的操作数集中在一个或几个标签中，人机界面只需使用少量的几个通信连接，便优化地获取了它所需要的通信数据，动态缓存区的更换也将更为便利和快速，人机界面访问的优化数据包，简单地用结构数据就实现了组合处理。
- 针对冗余数据交叉装载而创建　在冗余系统中，主控制器需要传送冗余数据到辅控

制器，每当主控制器运行的数据被刷新，即使没有变化，也要传送一次，挑选变化频度相同的数据组合成一个结构数据，将多次传送变为一次传送，可大量减少交叉装载的时间，从而优化冗余系统性能。

- 针对专门用途而创建　某些专用的通信模块，特别是第三方通信的模块，曾经是多种数据的交换，比如说组态数据、状态数据、输入数据、输出数据等，都可以将它们集合在一个结构数据中，并加以详细说明，过去需要一本厚厚的说明书对照操作来编写程序，现在由厂家提供一个或多个特定的结构数据就解决问题了。
- 由 Add_On 指令创建而生成　用户自定义指令 Add_On 在创建时，根据需求要建立各种类型的指令参数，这些指令参数将自动生成一个与 Add_On 指令同名的用户自定义结构数据。用于 Add_On 指令的自定义结构数据，跟标准指令的系统预定义结构数据在运用上几乎没有区别。

由于用户自定义结构数据是出于不同的目的建立的，很有可能同一操作数会出现在不同的结构数据标签中，比如有一个开关量操作数存在于作为控制对象而建立的结构数据标签中，同时这个开关量状态要被人机界面访问，为了组合优化通信，有可能这个开关量操作数被装入一个用于通信的结构数据标签中。如果你在一个控制器项目中，看到诸如名为 HMI 的例程，你完全有理由相信，例程中执行的操作都是将人机界面访问对象装入通信结构数据标签。

用户自定义结构必须是 32 位的基本数据单元的整倍数，每当创建一个子元素，系统都会进行比较，前面基本数据单元剩余的空间是否够用，如果不够，将开辟一个新的基本数据单元，即使是不同的数据类型集合在一起，系统还是会按字节紧凑地使用每一个空间。当然，子元素的开出先后顺序直接地影响了空间的紧凑与否，在必要的时候需对顺序进行调整。

定义数据的结构类型时，一定要紧凑地安排空间，如果留有多余的未使用空间，一旦这个数据类型被选用于标签，便会成倍地耗用内存空间。

总之，用户自定义数据类型的运用是灵活多变的，与编程处理的关系非常密切，不同的编程处理目的，创建的结构数据标签也不一样，在今后的编程训练中，我们会不断地接触到各种用法，届时再细细体会。

还有一种数据形式是数组。数组是同一数据类型连续分布的集合，既可以是基本数据类型组成，也可以是结构数据类型组成。数组也就是传统产品 PLC5 所说的文件，所以原 PLC5 指令系统的文件操作指令在 Logix 控制器的指令系统中统称为数组操作指令。

不同于文件的是数组可定义为 1 维、2 维和 3 维的数组，其中 1 维数组相当于原来的文件。各维数组中的元素个数基本不受限制，可以定义到任意大，直到内存不够使用为止。多维数组的数据存放形式，给可能存在的人机界面或二级计算机提供了模样完整的原始数据，为各种管理型数据库的数据处理奠定了良好的基础，尤其是集成架构系列的产品，更显出其优势和发展空间。

2.3　编程实验设备简介

本书选定 CompactLogix 控制器系统作为编程实验平台，CompactLogix 平台集中了 Logix

系列控制器的各种优势，CompactLogix 控制器不但具备了系列 Logix 控制器所具有的产品特征，较之 ControlLogix 控制器，结构更为紧凑，价格更为经济，是编程实验课程设备性价比颇高的最佳选择。

CompactLogix 控制器作为编程实验平台的优势在于：

- 硬件结构紧凑轻巧，可使用离散量和模拟量 I/O 混合型模块。
- 通用的以太网络易于与编程终端连接并和系统的网络搭接。
- 完整的指令系统功能齐全，4 种编程模式都能操作。
- 通用的编程软件用于组态 I/O 系统、建立数据标签和编写程序代码。
- 重量轻，体积小，便于携带和收纳。

如图 2-3 所示，是完成本书中所有的编程练习用到的一套编程实验设备系统。该设备系统简洁而不简单，它将有效地帮助我们完成各个章节中的编程实操训练。编程实验设备选用一个 CompactLogix 控制器、一个离散量 I/O 模块和一个模拟量 I/O 模块组合而成，设计合理的面板信号系统将提供 6 个开关量输入点、4 个开关量输出点、4 个模拟量输入通道和两个模拟量输出通道，这正是生产工艺过程中现场的四种常规信号。此外，每一个对应于面板信号的外接信号可为特定的程序测试提供所需要的仿真环境，能够逼真地显示现场场景，演示真实的运行过程。



图 2-3　编程实验设备系统

编程实验设备系统包括：

- 1769-L35E　具有以太网通信接口的 CompactLogix 控制器；
- 1769-IQ6XOW4　6 输入点和 4 输出点的离散量混合模块；
- 1769-IF4XOF2　4 输入通道和 2 输出通道的模拟量混合模块；
- 内嵌在编程设备中的以太网路由器　包含两个有线端口和 WIFI 无线端口；
- 编程终端　台式 PC 或笔记本电脑，安装有 RSLogix5000 编程软件。

1769-L35E 是 CompactLogix 中最具代表性的控制器模块。它保留了传统的串口，作为初始在线连接，用以完成初始通信组态。一个常规以太网通信端口，通常用作编程端口和对外通信端口。控制器的通信容量支持 100 个连接、32 个缓存式（Cached）连接、3 个非连接进入排队（Incoming），10 ~ 40 个非连接发送排队（Outgoing）。控制器面板上的以太网口支持

32 通信连接，使用时要留有余地，以便编程终端在线联机或其他非 I/O 设备的接入。控制器最多可创建 8 个任务，每个任务下可创建 32 个程序，每个程序下创建的例程个数不受限制。

1769-IQ6XOW4 是离散量模块。输入信号有汇回路和源回路之分，由于信号输入回路是晶体管的单向回路，当外部连接含源设备时，要注意外接 24V 电源的正负极性连接端是不同的。输出回路为继电器回路，可接 5 ~ 125V 直流回路，或接 5 ~ 265V 交流回路。可分别对控制器非正常工作状态组态输出，选择控制器编程模式下输出回路状态及模块与控制器发生通信故障时输出回路状态，这给系统被控设备的安全性提供了保证。

1769-IF4XOF2 是模拟量模块。输入通道可以是单端信号输入、双端信号输入，或单端信号和双端信号混合输入，取决于每个通道的接线方式，输入信号可以是 0 ~ 10V，也可以是 0 ~ 20mA，对信号具有固定的过滤作用，当输入信号变化过快时将削弱信号。输出通道可以是电压信号源，也可以是电流信号源。由组态模块产生的系统预定义结构数据标签包含组态数据、输入数据和输出数据。这个模拟量模块的 A/D 和 D/A 转换属于 8 位低精度转换，但是放置在 16 位寄存器的第 7 ~ 14 位，故可获得 0 ~ 31140 的较大数据范围，这样的原始数据一般要经过编程进行定标处理，模块本身并无数据定标处理能力。

此外，插放在模块组合的硬件系统左右侧的左端盖板 1769-ECL 和右端盖板 1769-ECR 也是必不可缺的，它们相当于并联在硬件系统背板两端的终端电阻。

内嵌在编程设备上的两个以太网有线端口可作为编程设备之间的连接端口，将编程设备搭接在一个以太网中，以完成通信指令的编程测试，也可用来连接作为编程终端的计算机。WIFI 无线端口则用来连接笔记本电脑，作为一套完整的编程设备系统，编程终端与编程设备相连，除了必要的电源线，没有任何连线，实验台面非常简洁干净。

最方便的编程终端是笔记本电脑。在微软平台的 Windows XP 操作系统运行环境中，安装编程软件 RSLogix5000 及连接软件 RSLinx。内置的以太网口作为外接端口，或利用 WIFI

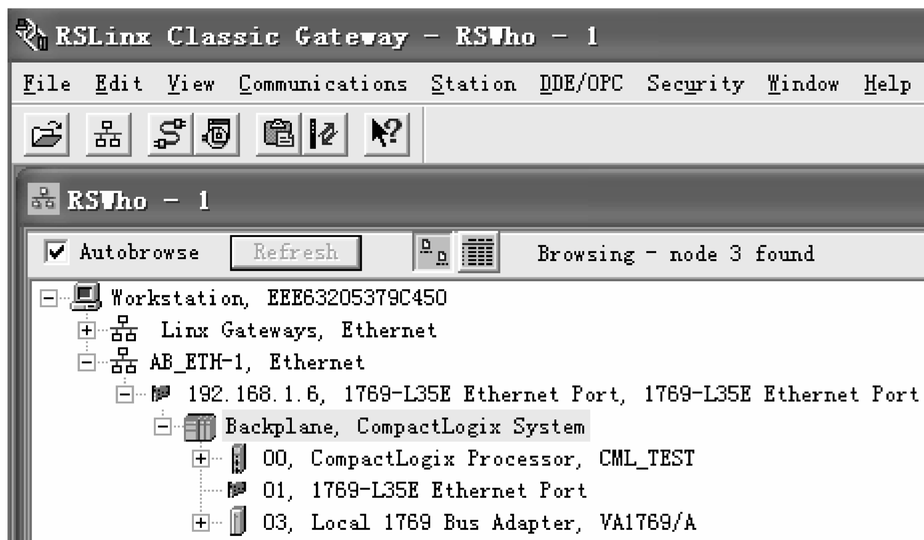


图 2-4　在线联机浏览

的无线连接端口接至编程实验设备的以太网路由器，以太网路由器有线端口则用网线直连至1769-L35E 以太网端口。连接软件 RSLinx 将组态以太网的连接驱动，以建立编程软件RSLogix5000 与控制器之间的在线联机，实现下载和上载控制器项目文件，在线测试和修改程序代码，在线监视和修改实时数据等。如图 2-4 所示。

2.4 控制器项目组态

在开始编程练习之前，我们先离线创建一个项目，作为编程的一个基本平台，等编制好要执行的程序代码后，再将项目文件下载到控制器，在线进行测试和调试。

创建一个名为 CML_Test 的新项目，选择 CompactLogix 控制器型号为 1769-L35E，选择版本 18，项目文件存放在路径 C：\ RSLogix5000 \ Projects 处，如图 2-5 所示。

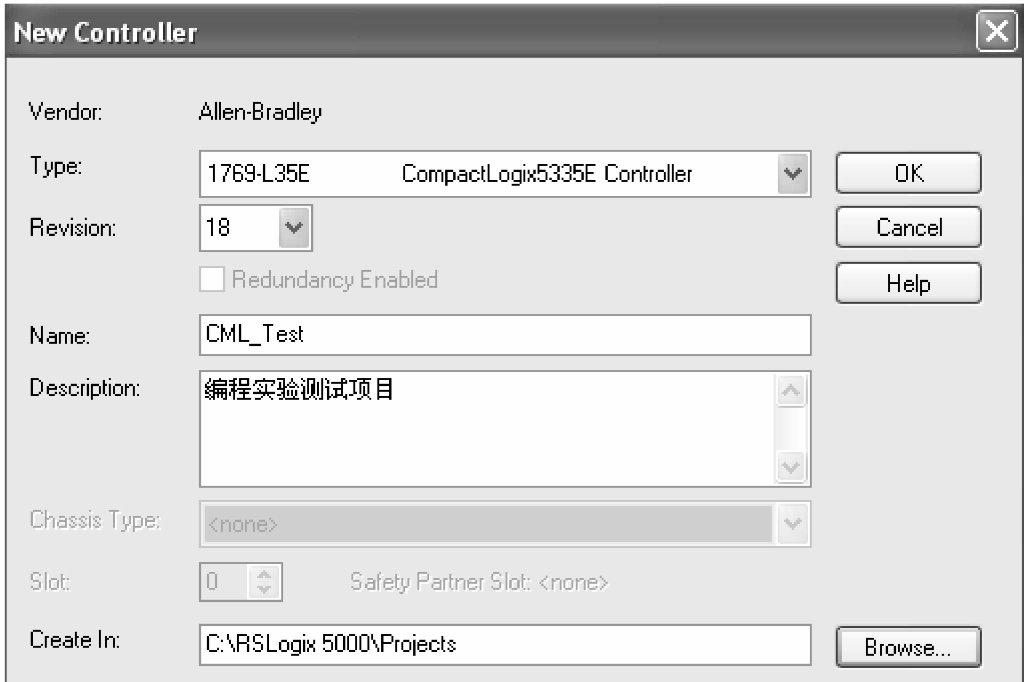


图 2-5 创建新项目

点击“OK”按钮，创建后的项目如图 2-6 所示。

I/O 组态树形分支下出现了 CompactLogix 系统的背板功能结构，项目被命名为 CML_Test 的 1769-L35E 控制器、本地以太网通信口以及本地的 CompactBus，这是最基本的一个本地 I/O 的 CompactLogix 系统结构，也是我们编程实验设备所选择的硬件系统结构。

于 CompactBus Local 下面创建本地离散量 I/O 混合模块，模块命名为 DInput_DOutput，位于 CompactBus Local 的 1 槽，如图 2-7 所示。

在连接页面 Connection，设定 RPI 时间为 5ms，如图 2-8 所示。屏蔽模块和运行模式下的连接故障引起的控制器主要故障，这两个选项都不勾选，在我们的程序测试中，暂时不涉及有关 I/O 模块的通信故障处理，如果选择该项可能导致停机。

创建并完成组态的 1769-IQ6XOW4 模块，在控制器数据区域产生离散量混合 I/O 模块结

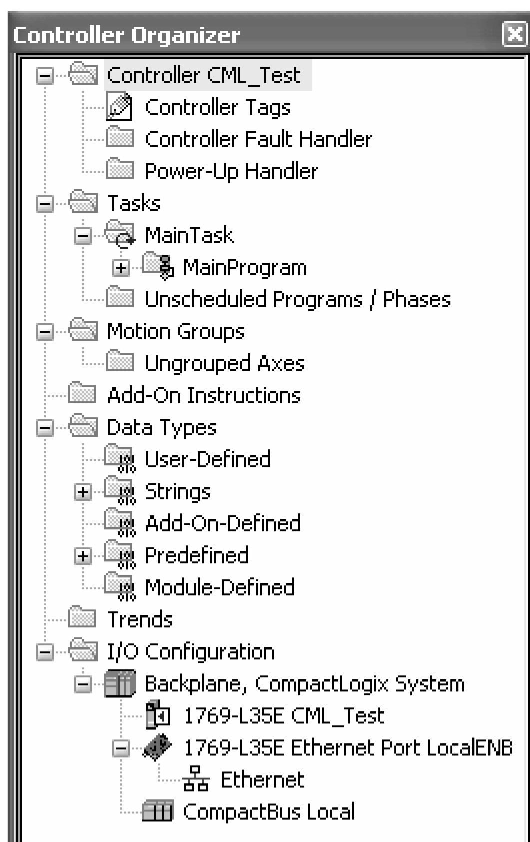


图 2-6　创建后的项目

构数据表，如图 2-9 所示。

模块组态数据说明：

- ProgToFaultEn　当控制器编程模式下模块通信故障时选择，设为 0 按照控制器编程模式的设定输出；设为 1 按照模块通信故障的设定输出。每个设置位对应相应的输出点。
- ProgMode　控制器编程模式下输出回路的状态，设置为 1 保持最后状态；设置为 0，用户定义状态，在 ProgValue 中设定。每个设置位对应相应的输出点。
- ProgValue　控制器编程模式下输出回路的用户定义状态，设为 0 或 1。每个设置位对应相应的输出点。
- FaultMode　模块通信故障输出回路的状态，设置为 1 保持最后状态；设置为 0，用户定义状态，在 FaultValue 中设定。每个设置位对应相应的输出点。
- FaultValue　模块通信故障输出回路的用户定义状态，设为 0 或 1。每个设置位对应相应的输出点。

模块输入数据说明：

- Fault　输入故障状态，只使用 0 ~ 5 位，6 ~ 15 位不使用。
- Data　输入数据，只使用 0 ~ 5 位，6 ~ 15 位不使用。
- ReadBack　从输出回路读回的回路状态，只使用 0 ~ 3 位，4 ~ 15 位不使用。

模块输出数据说明：

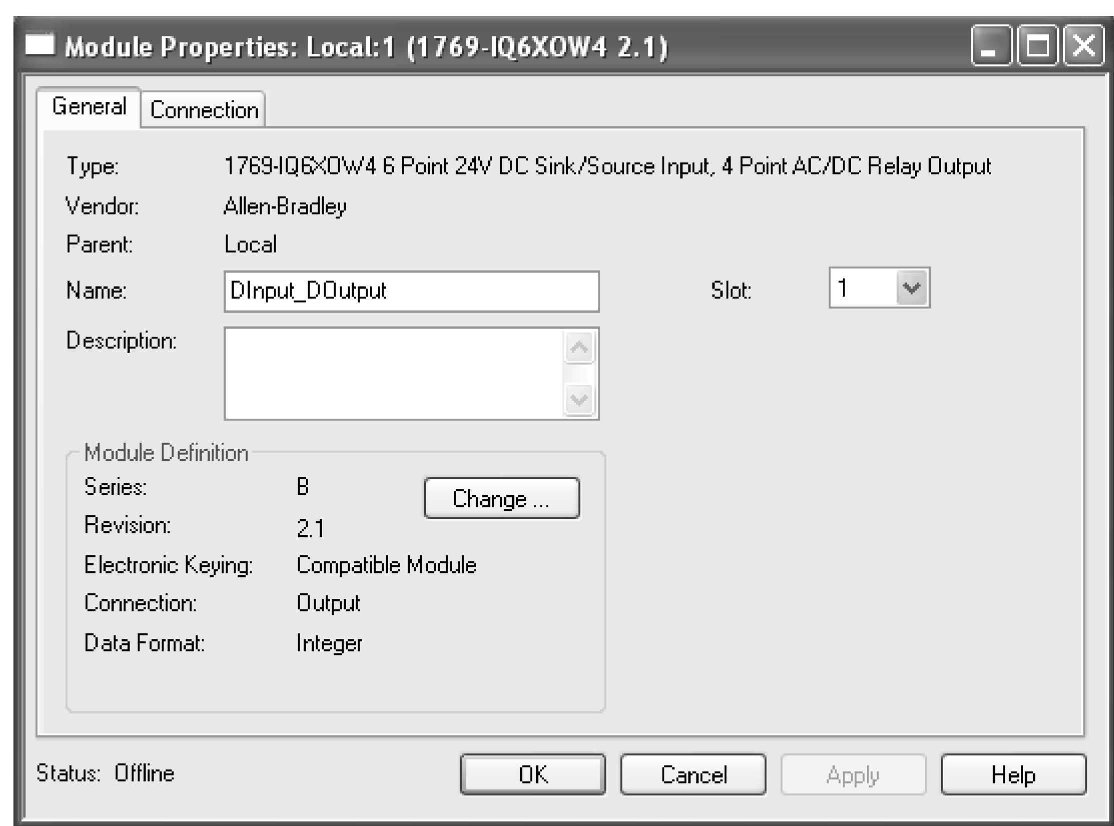


图 2-7　创建本地离散量 I/O 混合模块

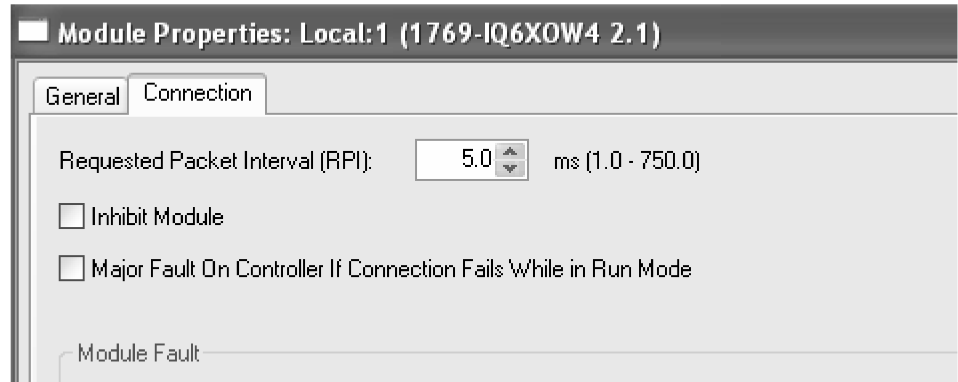


图 2-8　设定 RPI 时间

● Data　控制输出数据，只使用 0 ~ 3 位，4 ~ 15 位不使用。

值得提醒的是，1769-IQ6XOW4 模块原适用于 Micro1500 的控制器，组态方式是直接在数据表中完成，当我们在 RSLogix5000 的软件中组态，仍然要在数据表中设置，而不是在组态页面上设置并会自动传递到数据表中，所以要了解数据表中数据设置的含义。

离散量输入字的 0 ~ 5 位对应了编程实验设备面板上的旋钮 DI0、DI1、DI2、DI3、DI4 和 DI5；离散量输出字的 0 ~ 3 位对应了编程实验设备面板上的灯 DO0、DO1、DO2 和 DO3，如图 2-10 所示。输入信号和输出信号的每个端子的左侧为面板信号，可直接在面板上操作

⊟ Local:1:C	{...}
⊞ Local:1:C.Config	2#0000_0000_0000_0000
Local:1:C.ProgToFaultEn	0
⊞ Local:1:C.ProgMode	2#0000_0000
⊞ Local:1:C.ProgValue	2#0000_0000
⊞ Local:1:C.FaultMode	2#0000_0000
⊞ Local:1:C.FaultValue	2#0000_0000
⊟ Local:1:I	{...}
⊞ Local:1:I.Fault	2#0000_0000_0000_0000_0000_0000_0000_0000
⊞ Local:1:I.Data	2#0000_0000
⊞ Local:1:I.ReadBack	2#0000_0000
⊟ Local:1:O	{...}
⊞ Local:1:O.Data	2#0000_0000

图 2-9　离散量混合 I/O 模块结构数据表

和显示；右侧的插孔为外接信号，可通过插头连接外部的同类信号。

于 CompactBus Local 下面创建本地模拟量 I/O 混合模块，模块命名为 AInput_AOutput，位于 CompactBus Local 的 2 槽，如图 2-11 所示。

在连接页面 Connection，设定 RPI 时间为 5ms，如图 2-12 所示。屏蔽模块和运行模式下的连接故障引起的控制器主要故障，这两项都不勾选，在我们的程序测试中，暂时不涉及有关 I/O 模块的通信故障处理，如果选择该项可能导致停机。

如图 2-13 所示，在模拟量输入组态界面，选择 4 个通道都使能。这 4 个通道使能也可以通过程序对模块结构数据标签的子元素进行修改，当通道使能，模拟量输入数据被读入数据表中，供编程使用。

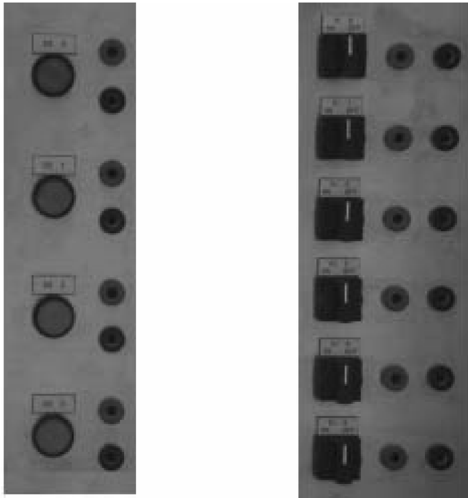


图 2-10　编程实验设备面板

这 4 个通道可接受 0～10V 的电压信号输入，或 0～20mA 的电流信号输入，经过 A/D 转换后，转换的数据范围为 0～31140。这里输入电压信号和输入电流信号的转化精度是不一样的。我们的实验设备将选用 0～10V 的电压信号作为输入信号。

由于 1769-IF4XOF2 模块本身不能完成数据定标工作，所读取的模拟量输入数据为 A/D 转换的直接结果，数据定标只能通过程序运算完成。本书的实例中，为方便起见，来自模拟量信号的 A/D 转换数据都编程换算为 0～10000 的数据定标范围。

如图 2-14 所示，在模拟量输出组态界面，选择两个通道都使能。这两个通道使能也可以通过程序对模块结构数据标签的子元素进行修改，当通道使能，程序控制的输出数据被送至模拟量输出。

这两个通道的数据输出范围为 0～31140，经过 D/A 转换后，可产生 0～10V 的电压信号

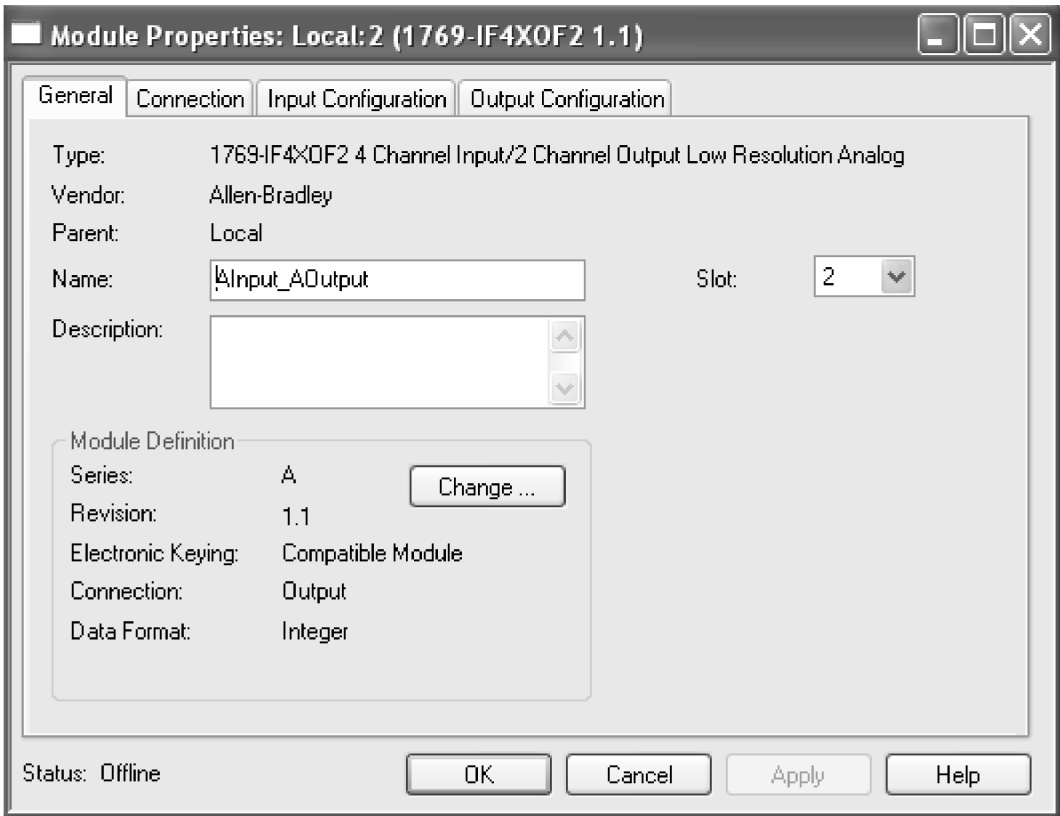


图 2-11　于 CompactBus 下面创建本地模拟量 I/O 混合模块

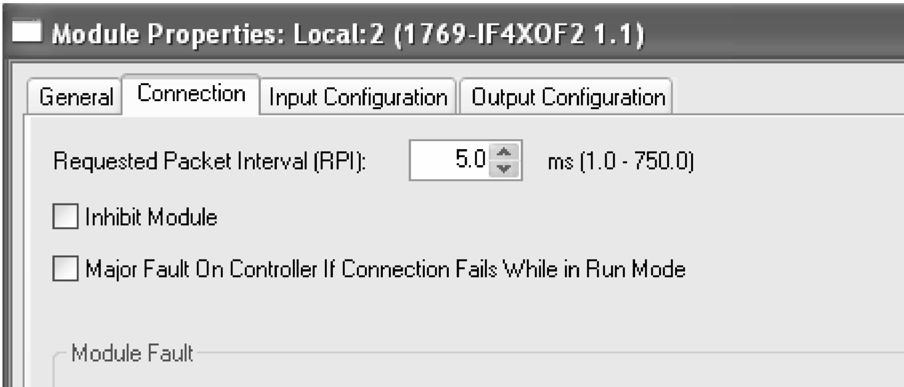


图 2-12　设定 RPI 时间

输出，或 0 ~ 20mA 的电流信号输出。这里输出电压信号和输出电流信号的转化精度是不一样的。我们的编程实验设备将选用 0 ~ 10V 的电压信号作为输出信号。

由于 1769-IF4XOF2 模块本身不能完成数据定标工作，控制器送到模块的模拟量输出数据将直接进行 D/A 转换，数据定标只能通过程序运算完成。本书的实例中，为方便起见，

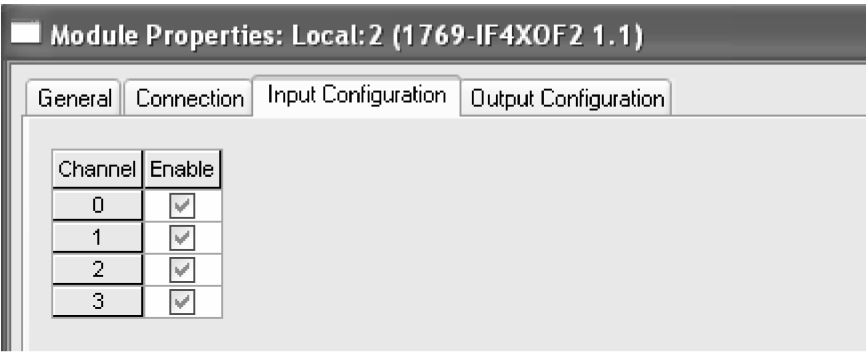


图 2-13　模拟量输入通道使能

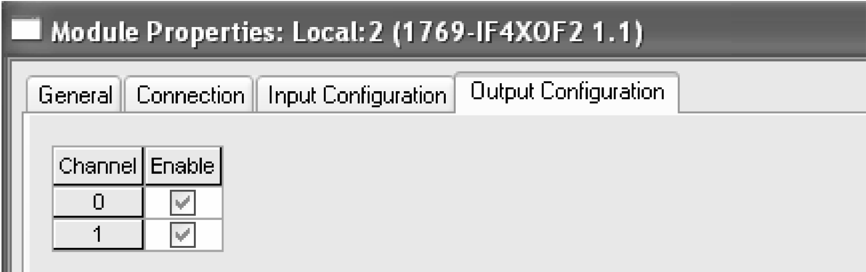


图 2-14　模拟量输出通道使能

送至模块 D/A 转换为模拟量信号的数据都编程换算为 0 ~ 10000 的数据定标范围。

创建并组态完成的 1769-IF4XOF2 模块，在控制器数据区域产生模拟量 I/O 混合模块结构数据表，如图 2-15 所示。

模块组态数据说明：

- Ch0ProgToFaultEn　模拟量输出通道 0 使能，当控制器编程模式下与模块通信故障时的选择，设为 0 按照控制器编程模式的设定输出；设为 1 按照模块通信故障的设定输出。
- Ch0ProgMode　模拟量输出通道 0 控制器编程模式设定使能。
- Ch0FaultMode　模拟量输出通道 0 控制器与模块通信故障模式设定使能。
- Ch0InputEn　模拟量输入通道 0 使能。
- Ch1InputEn　模拟量输入通道 1 使能。
- Ch2InputEn　模拟量输入通道 2 使能。
- Ch3InputEn　模拟量输入通道 3 使能。
- Ch1ProgToFaultEn　模拟量输出通道 1 使能，当控制器编程模式下模块通信故障时选择，设为 0 按照控制器编程模式的设定输出；设为 1 按照模块通信故障的设定输出。
- Ch1ProgMode　模拟量输出通道 1 控制器编程模式设定使能。
- Ch1FaultMode　模拟量输出通道 1 控制器与模块通信故障模式设定使能。
- Ch0OutputEn　模拟量输出通道 0 使能。
- Ch1OutputEn　模拟量输出通道 1 使能。
- Ch0FaultValue　设置模块通信故障时的模拟量输出通道 0 的输出值。
- Ch0ProgValue　设置控制器编程模式时的模拟量输出通道 0 的输出值。
- Ch1FaultValue　设置模块通信故障时的模拟量输出通道 1 的输出值。

⊟-Local:2:C	{...}		AB:1769_IF4XOF2:C:0
⊞-Local:2:C.Config0	2#0000_0000_1111...	Binary	INT
Local:2:C.Ch0ProgToFaultEn	0	Decimal	BOOL
Local:2:C.Ch0ProgMode	0	Decimal	BOOL
Local:2:C.Ch0FaultMode	0	Decimal	BOOL
Local:2:C.Ch0InputEn	1	Decimal	BOOL
Local:2:C.Ch1InputEn	1	Decimal	BOOL
Local:2:C.Ch2InputEn	1	Decimal	BOOL
Local:2:C.Ch3InputEn	1	Decimal	BOOL
⊞-Local:2:C.Config1	2#0000_0000_0011...	Binary	INT
Local:2:C.Ch1ProgToFaultEn	0	Decimal	BOOL
Local:2:C.Ch1ProgMode	0	Decimal	BOOL
Local:2:C.Ch1FaultMode	0	Decimal	BOOL
Local:2:C.Ch0OutputEn	1	Decimal	BOOL
Local:2:C.Ch1OutputEn	1	Decimal	BOOL
⊞-Local:2:C.Ch0FaultValue	0	Decimal	INT
⊞-Local:2:C.Ch0ProgValue	0	Decimal	INT
⊞-Local:2:C.Ch1FaultValue	0	Decimal	INT
⊞-Local:2:C.Ch1ProgValue	0	Decimal	INT
⊟-Local:2:I	{...}		AB:1769_IF4XOF2:I:0
⊞-Local:2:I.Fault	2#0000_0000_0000...	Binary	DINT
⊞-Local:2:I.Ch0Data	0	Decimal	INT
⊞-Local:2:I.Ch1Data	0	Decimal	INT
⊞-Local:2:I.Ch2Data	0	Decimal	INT
⊞-Local:2:I.Ch3Data	0	Decimal	INT
⊞-Local:2:I.InputRangeFlag	2#0000_0000_0000...	Binary	INT
Local:2:I.Ch0InputOverRan...	0	Decimal	BOOL
Local:2:I.Ch1InputOverRan...	0	Decimal	BOOL
Local:2:I.Ch2InputOverRan...	0	Decimal	BOOL
Local:2:I.Ch3InputOverRan...	0	Decimal	BOOL
⊞-Local:2:I.OutputRangeFlag	2#0000_0000_0000...	Binary	INT
Local:2:I.Ch0OutputOverR...	0	Decimal	BOOL
Local:2:I.Ch1OutputOverR...	0	Decimal	BOOL
Local:2:I.Ch0DataInvalid	0	Decimal	BOOL
Local:2:I.Ch1DataInvalid	0	Decimal	BOOL
⊞-Local:2:I.Ch0Readback	0	Decimal	INT
⊞-Local:2:I.Ch1Readback	0	Decimal	INT
⊟-Local:2:O	{...}		AB:1769_IF4XOF2:O:0
⊞-Local:2:O.Ch0Data	0	Decimal	INT
⊞-Local:2:O.Ch1Data	0	Decimal	INT

图 2-15　模拟量 I/O 混合模块结构数据表

- Ch1ProgValue　设置控制器编程模式时的模拟量输出通道 1 的输出值。

模块输入数据说明：

- Ch0Data　模拟量输入通道 0 数据。
- Ch1Data　模拟量输入通道 1 数据。
- Ch2Data　模拟量输入通道 2 数据。
- Ch3Data　模拟量输入通道 3 数据。
- Ch0InputOverRange　模拟量输入通道 0 信号超范围状态位。
- Ch1InputOverRange　模拟量输入通道 1 信号超范围状态位。
- Ch2InputOverRange　模拟量输入通道 2 信号超范围状态位。
- Ch3InputOverRange　模拟量输入通道 3 信号超范围状态位。
- Ch0OutputOverRange　模拟量输出通道 0 信号超范围状态位。
- Ch1OutputOverRange　模拟量输出通道 1 信号超范围状态位。
- Ch0OutputDataInvalid　模拟量输出通道 0 数据无效位。
- Ch0OutputDataInvalid　模拟量输出通道 1 数据无效位。
- Ch0ReadBack　模拟量输出通道 0 数据读回。
- Ch1ReadBack　模拟量输出通道 1 数据读回。

模块输出数据说明：

- Ch0Data　模拟量输出通道 0 数据。
- Ch0Data　模拟量输出通道 1 数据。

值得提醒的是，1769-IF4XOF2 模块原适用于 Micro1500 的控制器，组态方式是直接在数据表中进行，当我们在 RSLogix5000 的软件中组态，除了通道使能，大部分仍然要在数据表中设置，而不是在组态页面上设置，所以要了解数据表中数据设置的含义。

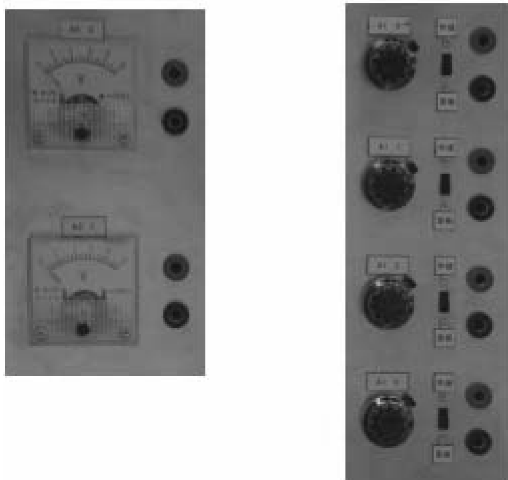


图 2-16　编程实验设备面板

在数据表中模拟量输入通道 Ch0Data、Ch1Data、Ch2Data 和 Ch3Data 所显示的数据，对应了编程实验设备面板上旋钮的 AI0、AI1、AI2 和 AI3 从 0 ~ 10V 的输入信号范围；数据表中模拟量输出通道 Ch0Data 和 Ch1Data 所显示的数据，对应了编程实验设备面板上电压表的 AO0 和 AO1 从 0 ~ 10V 的输出信号显示范围，如图 2-16 所示。输入信号和输出信号的每个通道的左侧为面板信号，可直接在面板上操作和显示；右侧的插孔为外接信号，可通过插头连接外部的同类信号。

在后面各个章节的编程实例中，这些离散量和模拟量的 I/O 数据会被重复使用，由于 I/O 通道的资源有限，被测试的例程将不会同时使用 I/O 数据，以确保外部数据的使用不会发生冲突。

创建项目时已自动生成连续任务，在连续任务下已自动生成程序 MainProgram，在程序

下已自动生成 MainRoutine，如图 2-17 所示。

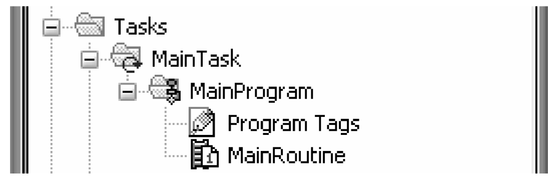


图 2-17　项目结构

打开例程 MainRoutine，如图 2-18 所示，将在梯形图中编写梯级逻辑，并将项目文件下载到控制器。在线运行测试，并可对编写的执行代码进行调试或修改。

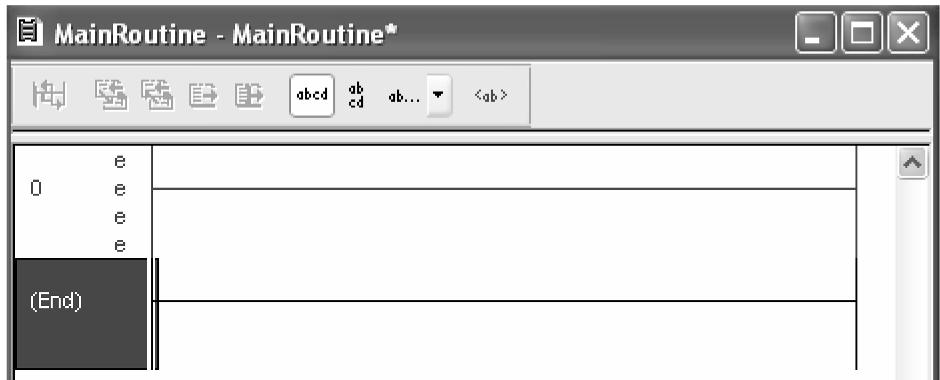


图 2-18　打开例程 MainRoutine

以上，因为创建了从属于控制器的 I/O 模块，由此在控制器的数据区域中产生了离散量 I/O 模块和模拟量 I/O 模块的结构数据标签，关于其他标签的创建，我们将在例程编写的时候针对编程需求逐步创建。

第 3 章

梯形图编程基础

尽管 PAC 集成了多种控制功能，具有多种编程模式，就目前来说，最常用的仍然是时序逻辑控制，也即 PLC 的应用。本书将基于 PAC 的指令系统介绍一般的编程方法，特别是常用的梯形图编程，用以完成时序逻辑的控制，作为编程能力的训练，学习梯形图编程的确是非常有效的方法，也是 PAC 编程最基本的训练。

梯形图表达形式来自于电工习惯的电路结构，它直观地表达了指令之间的逻辑关系，控制器运行后，梯形图使能所显示的工作有效的图形，犹如上电后的元器件通断状态的电路，使能有效的逻辑指令，犹如通电的元器件。事实上，最初的梯形图基本指令就是元器件的代

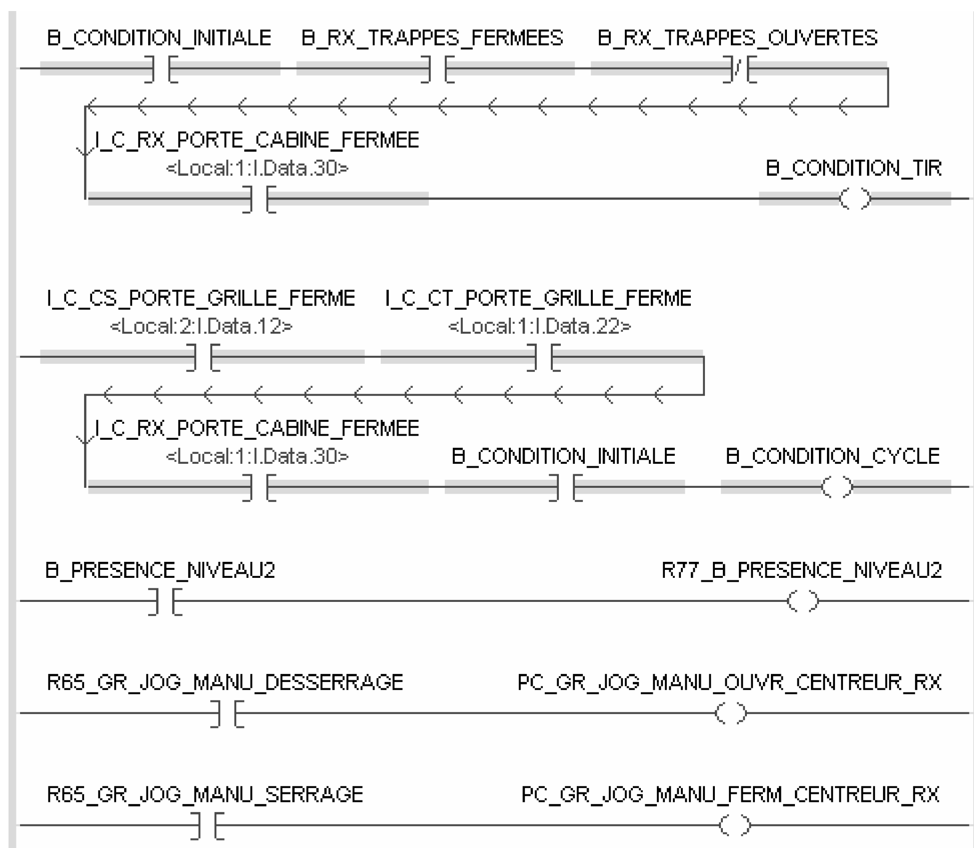


图 3-1　实例摘录

称，如继电器指令，迄今为止，继电器指令仍然是梯形图中出现得最为频繁的指令，尽管有时并非跟现场硬件设备一一对应，仅仅是表达逻辑关系中间变量的一个内部线圈，它是如此清晰地表达了某种逻辑关系，令人一目了然。现在人们更概括地将这种表达逻辑关系的指令称为位指令，当然，这样的指令所使用的就是位地址，一个布尔量的表达。如图 3-1 所示是从实际运行程序中摘录的一部分，你甚至可以认为这是电路元件图的软件化显示。

梯形图顾名思义形如楼梯，一级一级的梯阶，拾阶而上行如进程，每个梯级的指令，构成了执行的逻辑，加以时间的顺序，人称时序逻辑控制。程序执行的动作就是梯形图按顺序的扫描，梯形图扫描遵循从上到下，从左到右的规则，从上到下顺序地扫描每个梯级，从左到右顺序地扫描每条指令。显然，梯级前后顺序的位置决定了指令执行的先后，从而决定了控制的因果关系，前一级的控制结果，也许就是后一级的控制条件，如此递进有序不紊。对于一个完成指令动作的梯级来说，在梯形图中摆放的位置是颇为讲究的，这是在编写程序时要特别注意的地方。有时为了得到正确的结果，我们会不厌其烦地更换梯级的位置，或者在调试的时候改变梯级的位置，在今后的编程中，我们将获得这样的经验。

在梯形图的每个梯级中，指令又是如何执行的呢？从左到右的扫描规则是如何地安排了指令的动作呢？如果希望编写的程序执行代码能够精准无误地执行，完成我们设定的逻辑控制，得到我们预期的逻辑结果，就必须对梯级中指令的执行过程有着充分的了解和把握。以下要谈到的梯形图编程基础，就是详细地分析一个梯级的结构和指令的执行条件。

3.1　梯形图的梯级结构

一般地，梯形图中每个梯级的结构由输入指令和输出指令两部分组成，在编写梯形图例程结构的时候，控制器操作系统会识别指令的类型，并自动地将输入指令放置在左边，将输出指令放置在右边，如图 3-2 所示。输入指令和输出指令并非数据的输入和数据的输出，而是直译过来，原意大约指的是位于梯级的位置。

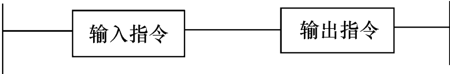


图 3-2　梯级结构

输入指令决定了梯级条件，输出指令则按照梯级条件执行。输出指令用来完成控制的动作，任何一个梯级，可以没有输入指令，但必须有输出指令，没有输出指令的梯级在编辑时是不能被接受，属于语法错误。

对于需要依赖梯级条件的持续存在才能执行的输出指令，它所在的梯级可以没有输入指令，不妨认为是无条件或梯级条件永远成立，该条输出指令将在例程的每个扫描周期都会执行；对于需要梯级条件跳变才能执行的输出指令，它所在的梯级没有输入指令，可能令输出指令一直被使能，如果后面的梯级没有做本条指令使能复位的特殊处理，这条指令在执行一次动作后，将不再执行。

1. 输入指令

输入指令是决定梯级条件是否成立的指令，只有当梯级条件成立，后面的输出指令才有可能执行，输入指令通常是以下三类指令：

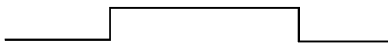
- 位指令　位指令输入只有常开 XIO 和常闭 XIC 两条指令，却是在梯形图中用得最多的，根据生产控制过程的开关量设备状态，或某些中间结果位，编制与、或、非等逻辑关系的逻辑结构，其逻辑运算结果必为逻辑，是或不是，从而决定梯级条件是否成立。
- 比较指令　这是一组用于范围判断的比较指令，用大小相等关系进行比较，也可以对限定的数值范围进行判定，甚至直接对表达式运算的结果进行比较，其比较结果必为逻辑结果，是或不是，从而决定梯级条件是否成立。
- 检测或诊断输入指令　限于两条特殊的输入指令，顺序器输入指令 SQI 和数据传送指令 DTR 对操作对象进行较为复杂的检测和诊断，这是一种按照事先设定对照参数的检测和诊断，其对照结果的相同或不同必为某个逻辑结果，是或不是，从而决定梯级条件是否成立。

显然，以上三种情况，所获得的结果都是逻辑的结果，逻辑的结果只有是或不是，也即梯级条件成立或不成立，以此来决定梯级条件。

控制器例程的执行过程是按照梯级顺序对每个梯级扫描，首先对梯级的输入指令扫描，判断梯级条件是否成立，梯级条件不成立，离开此梯级，继续扫描下一个梯级；梯级条件成立，则执行后面的输出指令，实施真正的操作动作。梯级条件仅仅是决定要不要实施输出指令的操作动作，梯级条件的存在有两种情形，如图 3-3 所示。

梯级条件的存续时间，直接影响输出指令的执行，不同的输出指令对梯级条件的要求是不一样的，有的输出指令依赖梯级条件的存在持续使能，有的输出指令需要梯级条件前沿触发使能，指令被使能将会如何动作，都是我们要弄清楚的。有时我们希望输出指令能够连续不断地执行，有时我们希望输出指令只被执行一次，梯级条件的怎样给予，是编写输出指令的重要依据，它将确定输入指令和输出指令配合使用的关系，不合适的输入指令和输出指令的搭配关系，可能会得出编程者所不希望的结果。

梯级条件持续一段时间，逻辑上表现为宽脉冲



梯级条件出现瞬间时间，逻辑上表现为窄脉冲



图 3-3　梯级条件存在的两种情形

2. 输出指令

输出指令才是完成执行动作的实质性操作，它将根据梯级条件来执行，不同的输出指令，对梯级条件的要求是不一样的，按照不同的梯级条件而引起的执行动作，通常分类为非保持型输出指令和保持型输出指令：

- 非保持型输出指令　当例程被扫描时，在梯级条件持续期间执行操作，梯级条件消失后便停止执行操作，最具代表性的是计时器指令，运算、转换和传送指令通常都是非保持型输出指令，如下所列：

-计时器指令 TON、TOF 和 RTO；
-非保持型输出位指令 OTE；
-传送指令 MOV、MVM 和 BTD；
-转换指令 TOD、FRD 和 DEG、RAD；
-算逻运算指令 ADD、SUB、MUL、DIV、CPT 和 AND、OR、XOR、NOT；
-拷贝指令 COP 和 CPS；

-清除指令 CLR、充填指令 FLL。

● 保持型输出指令　当例程被扫描时，在梯级条件跳变时执行操作，依靠梯级条件的前沿触发引起执行动作，最具代表性的是计数器指令。寄存器操作指令或数组操作指令通常都是保持型输出指令，如下所列：

-计数器指令 CTU 和 CTD；

-位锁存指令 OTL 和位解锁指令 OUT；

-寄存器移位指令 BSL 和 BSR；

-数组堆栈操作指令 FFL、FFU 和 LFL、LFU；

-顺序器输出指令 SQO 和顺序器装载指令 SQL；

-数组操作指令 FAL。

每当编写一条输出指令时，一定要认真研究这条输出指令对梯级条件的要求，不同的输出指令对梯级条件的要求是不同的，只有当输出指令获得了必需的梯级条件（持续或跳变），才能正确地执行，达到编程预期的结果。每个梯级、输出指令与梯级条件都有匹配的关系，输入指令就要提供正确匹配关系的保证。

3.2　梯级的预扫描和后扫描

要正确地编制程序，执行代码的确能像我们预期的那样动作，产生我们所需要的控制结果，我们常常会分析梯级中指令被扫描时的动作，但是在非正常扫描时，系统将对指令参数的数据作出怎样的处理呢，我们必须了解系统预扫描和后扫描的作用。

并非只有例程在正常扫描时，输出指令执行才会有目标地址的数据结果，还有预扫描和后扫描的作用也会影响到输出指令的目标地址的数据状态。预扫描的作用通常比较短促，常常是一闪而过，复位的情形比较明显，后扫描的作用常常是持续着的，在一些暂停的场合或未激活的状态都会表现出来，甚至会对输出控制的外部设备产生影响。由于预扫描和后扫描对我们编制的执行逻辑结果有一定的影响，所以在编写输出指令的执行动作时，必须要考虑这两种情况，否则你会觉得输出指令的表现有时不可思议。

预扫描和后扫描的动作一般不会被初学者所注意，只有当现场调试例程有了一定的经验后，才会注意到预扫描和后扫描对例程中输出指令施加的影响。预扫描和后扫描正是为了对数据进行预定或善后处理，属于系统介入的一种特定操作，这个系统介入的特定操作往往是用户应用中所需要而无法通过梯级逻辑执行实现的动作，其实我们一直乐在其中而浑然不觉。一个典型的例子就是计时器，我们可以看到控制器停机时计时器的累加值 ACC 会停止在某个数据，一旦控制器开始运行，预扫描令所有的非保持型的数据复位，计时器的 ACC 复位后开始重新计时，我们习惯了这样，以为平常，殊不知这就是预扫描的作用，可帮助我们重新开始。话又说回来，如果你的计时器在控制器暂停运行后重新启动，想要保持原有的累加值继续累加，你才会注意到原来累加值在开始运行时是会被复位的，于是就要设法来避免计时器的累加值复位，比如采用保持型的计时器来计时。

指令集中每一条指令的介绍，都列表说明了预扫描和后扫描对指令的作用，这点其实至关重要，却总是被人忽视，对此较为深刻的理解恐怕要到后面的章节，即用户自定义指令的编程训练中，大家自已来编写预扫描和后扫描处理之时方能体会。

简而言之，预扫描和后扫描是有别于常规梯级扫描的特殊处理，对所有例程的所有梯级实施一遍特殊的扫描，并进行特殊的处理。

预扫描是在例程进入正式运行之前完成的一次特殊操作，预扫描要做的事情是：

- 预扫描所有的主例程；
- 预扫描所有的设备阶段（Phase Manager）的状态例程；
- 预扫描所有的设备阶段（Phase Manager）的预状态例程；
- 预扫描程序和设备阶段（Phase Manager）的所有的子例程（不会重复预扫描）；
- 预扫描所有的 FOR 指令调用的子例程；
- 不依照跳转指令指向的顺序；
- 按预扫描方式执行每一条指令（指令集有详细说明）；
- 将所有的非保持型指令复位；
- 不刷新输入量；
- 不送出输出量。

如下两种情况不会做预扫描：

- 控制器进入运行时，程序或设备阶段（Phase Manager）为非预定性的；
- 控制器运行时，程序或设备阶段（Phase Manager）从非预定型转变为预定性的。

预扫描复位所有的非保持指令的操作数，例如 OTE 指令，非保持型计时器指令 TON 和 TOF 的累加值 ACC 等都会被复位为 0，但立刻就进入正常扫描状态。

预扫描最容易被观察到的是 TON 指令的累加值复位，当控制器工作状态从运行转向程序状态时，可以看到所有计时器的累加值都停留在某个值，保存的离线程序也是可以看到这样的数据状态，但是一旦开始运行，可以看到所有的非保持型的计时器的累加值全部复位为 0，重新开始计时。OTE 指令的状态是不明显的，尽管预扫描将指令结果已经复位，第一次正常扫描时，梯级条件立刻就决定了它的输出状态，预扫描的状态难以察觉。

控制器系统在某些例程或某段梯级逻辑从激活状态转入非激活状态时，要做的善后特殊处理被称为后扫描，如下几种情况要进行后扫描：

- MCR 指令结束区域控制时；
- 设备阶段（Phase Manager）的状态例程转向下一个预状态时；
- SFC 步的转换条件成立离开激活步时（如果设定）。

以上三种情况都是例程或某段梯级逻辑从激活状态转为非激活状态时，离开了正常的扫描处理，后扫描所做的事情是：

- 后扫描条件不成立的 MCR 区域；
- 后扫描所有的未激活的设备阶段（Phase Manager）状态例程；
- 后扫描所有的未激活的 SFC 的步（如果设定）；
- 按后扫描方式执行每一条指令（指令集有详细的说明）；
- 每个梯级仍然被扫描，但梯级条件永远不成立；
- 所有的非保持型指令都被复位；
- 刷新输入量但可能被梯级条件否认；
- 送出输出量并将起作用。

后扫描将静止例程或某段梯级逻辑中的每一个梯级，令所有的梯级条件不成立，故后扫

描将复位所有的非保持指令的操作数，例如 OTE 指令，非保持型计时器指令 TON 和 TOF 的累加值 ACC 等都会被复位为 0，这个复位情况会一直维持着，在梯级逻辑的整个非激活状态期间。

要特别留心语句编程的赋值语句，因为它不仅仅针对布尔量，数据量也会作用，语句编程的赋值语句分为保持型和非保持型两种，非保持型赋值在后扫描时将被复位或清除。

后扫描是系统提供给我们的一种特殊的功能，非常有用，有助于例程或梯级逻辑离开了激活的状态后，对目标数据所做的善后工作，因为例程或梯级逻辑停止扫描之后，就再也没有机会去处理离开之后希望数据所保持的状态，当被控制的是输出信号时，对外部被控制设备的影响尤其大，因为后扫描要送出输出数据，决定了设备状态，通常希望是关闭设备，即输出数据复位。

例如一个电动机的运转，在控制它的梯级处在非激活状态时，我们希望它自动地停止，而不需要额外地处理。这时使用 OTE 指令，不管当时的梯级条件如何，只要当前的梯级逻辑处于未激活状态，这个 OTE 指令会因为后扫描而复位电动机的控制，并且后扫描的输出是被送出并起作用的，这就是为这个电动机的控制选择非保持型输出指令 OTE 的理由。另外一种相反的情形，例如一个泵在控制它的梯级处于非激活状态时，仍然要维持它的运转，这时它的输出指令要考虑使用锁存输出指令 OTL，这个保持型的指令是不会因为后扫描而改变输出状态的。

尽管后扫描和预扫描一样，都会令非保持型的指令复位，但要注意后扫描与预扫描是有区别的。后扫描非激活区域的梯级静止活动是持续的，在非激活时段是一直作用着的，预扫描只有一次复位动作，之后马上被正常扫描赋予逻辑决定的状态。后扫描将送出输出量作用于输出点，而预扫描不会送出输出量，即使预扫描瞬间的复位，立刻又回到置位的状态，也不会造成外部输出点的扰动。不要轻视这个看似与编程无关的话题，在实际调试或运行中，恰恰是这些因素常常困扰我们的编程人员，在我们开始编程之前，就应该了解系统对程序介入的动作或作用，在编写逻辑的时候就考虑数据在不同情形下的状态，并决定如何处理，选择怎样的指令。

3.3 养成标准化编程的习惯

下面我们将要讨论如何运用指令编出令人满意的程序。毫无疑问，每一个编程的项目开发人员，都希望自己能编出好的程序，什么是好的程序？尽情地施展聪明才智，别出心裁，标新立异、奇思妙想、独树一帜、与众不同就是好的程序吗？我恐怕不能苟同。我认为好的程序应该具备如下几点：

- 严密性　逻辑严谨，执行准确，绝无疏漏，这点应该是共识，这个严密不仅有控制进程的严密，还有例程调用的严密，指令执行的严密。严密性跟编程之前的规划也是有关的，紧扣生产过程进程，分析控制对象的动作，仔细地规划项目的程序结构，安排执行的顺序，这都是保证严密的基础。
- 正确性　毋庸置疑的是控制逻辑处理的正确，针对控制对象的逻辑关系选对指令。正确地使用指令是非常重要的，什么情况该用什么指令，正确地理解指令执行的过程和正确设置参数，给予输出指令的梯级条件是否正确，预扫描和后扫描的影响如

何，这些都是需要考虑的，尽量地避免指令或例程的陷阱。

- 对称性　现场很多控制对象的活动具有对称性，尤其是一些具有互锁关系的动作，对应编写的逻辑处理也应具有对称性，例如几个互锁关系的同类控制动作，除了梯级条件和数据状态不同，梯级的逻辑结构应该是一样的，尤其是规律性的交替执行。对称动作采用不对称的逻辑执行，本身的严密性就值得怀疑，对称也是判断编程严密性的一个准则。
- 规律性　控制过程的动作大都具有规律，基于分析控制过程的功能框图，在编写的例程中要表现出这种规律，哪些是常规的主流执行动作，哪些是重复执行动作，哪些是条件式的调用动作，哪些是设备之间的约束关系以及生产进程的显示，都要有脉络清晰、明了的体现。
- 可读性　程序是写给别人看的，别人很容易读明白，搁置很久自己也能很快读明白，当时的得意之作，不要过一阵连自己也读不懂了。思路要清晰，表达要清楚，这点可以参照写文章的心得，有中心、有层次、有重点、有排比、有修饰，规律和对称的编程自然是可读性强的。当然，每个梯级逻辑或操作数的文字说明是必不可少的，这将帮助我们理解过程处理和编程思想，说明有时比梯级逻辑本身更重要。
- 标准化　对待同一个需求控制的处理，可以用指令功能解决的，不要用技巧编程去解决，因为指令功能是共性的，如何设置参数，如何运行指令，都有固定的模式，大家具有共识；编程技巧有时是很个性化的，即便构思非常精巧，但别人很难把握思路，特别是后期的维护人员不容易读懂程序。尤其 PAC 更是提供了标准化编程的平台，即使是新手也能编写出规范的程序。

以上几点最能体现好程序的是标准化，只有标准化的编程最容易达到严密、正确、对称、规律、可读，标准化往往是长期经验积累的最终形式，PLC 的发展历史就是一个沿标准化方向发展的过程，直接用于编程的指令系统表现则尤为明显。在 PAC 系统中提供各种编程方式，更是多方地提供了标准化的编程平台，这对当今项目开发所追求的短工期、高效率、低成本的目标，具有十分重大的意义，并在实践中得到认可。此外，标准化的编程还在项目开发者和现场维护人员之间达成了某些共识，使后者更容易解读程序和查找故障，这在生产实际中十分有效，尤其是某些行业，本来就有一些共同遵守的规则，更是要通过标准的程序来体现。

自从计算机处理模式的 PLC 进入工业自动化控制系统，将硬件的信号处理转为数字处理以来，针对工业控制自动化的需求，PLC 处理的计算机系统的控制性能也在不断地提高，就信息处理能力的提高而言，主要表现在指令系统的增加和丰富，指令的增多和指令功能的增强。在早期初级阶段的 PLC，由于指令功能较弱，在面对一些常见的需求处理时，需要更多的编程技巧去解决，当满足需求的这种技巧性处理的编程成为工程技术人员的一种共识，就有了变成指令功能的需求，如果将这些技巧处理组合为一个宏，不就是新的指令诞生了吗。当指令系统提升推出一些新功能的指令，从来没有突兀地诞生一条指令，一般都是跟需求和发展有关的进化。

这里我们用一条大家熟知的 ONS 指令为例来了解这个发展过程。

在梯形图的逻辑处理中，往往有这样的需求，对某些指令的执行，需要严格地确保只能执行一次；或者某些指令只需或只能在前沿触发，而此时梯级条件能给出的却是一个持续一

段时间的宽脉冲，在这段时间这条梯级可能经历了几百次的扫描，而我们想控制的是这个梯级的输出指令只能执行一次。显然，这时的需求是令这个宽脉冲变为一次扫描有效的窄脉冲。这就是硬件中经常被称为微分电路的一种需求转化而来的，在硬件中是靠电容元件的充放电来实现的，在软件中实现却颇费一番周折。

初期的 PLC2 没有 ONS 指令和 OSR/OSF 指令，每当遇到这样的需求，不得不编写一段梯级，让这段宽脉冲经过特殊处理后输出，成为一个保持一个扫描周期的窄脉冲，并应用于被需求的梯级条件。其编写的梯级如图 3-4 所示，这是一个梯级条件前沿触发产生窄脉冲的范例。

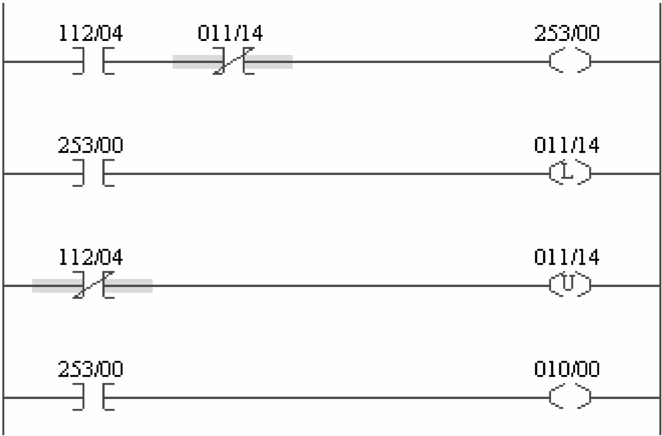


图 3-4　产生窄脉冲梯级的编写

显然，在今天看来，这是一段晦涩难懂的梯级逻辑编程。这段梯级逻辑编写的技巧，曾被作为典范拿来训练新的编程人员，让他们一读到这段逻辑处理，就认定前面 112/04 的位地址所产生的宽脉冲经过这段逻辑的处理，后面输出位 010/00 的结果是他们想要的前沿触发窄脉冲，这个位的置位状态能够维持一个扫描周期，可以使用这个位地址，获得一个周期扫描有效的脉冲作为所需梯级条件。还有一段相似的梯级逻辑得到的则是后沿触发窄脉冲，这里就不再列举了。

这种处理方式的不便是显而易见的，解读程序的人要善于将这一段插曲从动作进程主流梯级逻辑中择出，火眼金睛地将它译读为一个宽脉冲处理成的一个窄脉冲，只会保持一个扫描周期，并对这个窄脉冲的条件动作予以理会。要知道，这样的插曲在梯级逻辑处理中是不断地出现，甚至多到干扰人的视线。但是在没有一个特定的指令提供给我们使用于这种需求时，这实属无奈之举。

PLC5/SLC500 系列的处理器问世的时候，ONS 作为一条常用的指令，赫然出现在指令系统中，后来在 PLC5 增强型又推出了更详细的 OSR/OSF 指令。在宽脉冲的梯级条件，ONS 指令紧随其后，从而限制梯级条件改为了窄脉冲，直观而明了。编程人员要了解的只是如何使用这条指令，ONS 指令与前面梯级条件的关系如何，它的存储位是怎样作用的，什么情况下存储位被复位等。现在的工程技术人员大约难以体会到，当时的编程人员是如何大大地松了一口气，总算可以直接使用而无须拐弯抹角地去得到一个触发脉冲了。我们不难想象，ONS 指令不过就是刚才那段晦涩难懂的梯级逻辑的一个宏的集合罢了。

功能强大的指令系统的确是可以让人们免去冥思苦想地去寻求解决方案，也就是所谓的

技巧性处理，而直接运用了指令的功能，这些指令的功能往往源自于技巧性处理，由长期积累而成为共识的技巧性处理。特别是由经验积累而进化的指令功能，无疑最让新手受益，也是避免陷阱和出错的有力措施，这几乎成了指令系统性能提升的追求目标。所谓指令功能，也就是为解决需求的集众人智慧和长期经验的结晶。

面对同一个需求处理，我们可以看到两种编程处理情况：

技巧处理，可能是非常精妙的技巧，让人拍案叫绝；也可能让人百思不得其解，经过解释才让人明白；或者广泛流传的一种技巧，让人们达到了共识，如自保持位逻辑的编程方法。不管是什么情况，技巧是个性化的处理，难以作为标准。

功能处理，根据指令的功能和固有的指令编程模式，让解读程序的人很快地理解处理的目的和结果。这是通用的，所有的人都容易明白，指令功能是共性化的处理，标准的做法，但是需要学习指令的运用方法。

编程的一般原则是，能用指令功能处理的，尽可能不用技巧处理。道理是显而易见的，只有共性的东西才是可以共识的，容易共用的。毕竟一个程序项目是大家共有的，具有从开发人员移交到维护人员的延续性，要让更多的人更容易参与项目。

我们在学习编程中，一定要研究指令的功能和相应的运用方法，用以编写标准的，易读的程序，这就是编程训练的目的之一。所以，对控制器指令系统中的每一条指令给予充分的了解（参数的含义，指令的动作，执行的条件），才能准确无误地使用指令。尽可能地使用指令功能处理问题，尽量避免使用技巧编程处理。在后续的章节中，每类指令的编程训练，我们将进行对比，明了标准编程的重要性。

此外，养成良好的编程习惯也是至关重要的，记住某些指令的使用惯例和典型处理，这是前辈经验的流传，良好的编程习惯可以帮助你避免落入一些程序运行的陷阱，以免在调试系统时为某些逻辑处理结果感到困惑。良好的编程习惯可以让你对自己编写的程序具有自信，准确坚定地排除不可能情况，不会盲目置疑而耽误调试的时间。良好的编程习惯是经历了考验的工作方式，保持它可以减少错误的产生，加快编程速度，提高工作效率。

编程序就跟写文章一样，你既可以写成一篇随心所欲的散文，也可以写成一篇思维严密的论文。读散文每个人都可以有不同的理解和感受，甚至得出不同的结论；论文却可以让人准确地理解作者所要表达的内容，其论点明确、逻辑清晰、层次分明、结构严谨、论述清楚。把程序编得像一篇论文，应该就是编写好的程序的基本要求吧。

编程序跟写文章有许多相似之处。写文章，有中心思想，段落大意，在阐明一件事情时，你会围绕这件事情在一个文章段落里进行描述；编程序，有核心控制，运行进程，面对一个控制对象，在一段梯级逻辑中，编辑相关的逻辑关系和工作状态处理。写文章，你掌握的词汇越多，遣词造句的能力就越强，表述事情就越清晰准确；编程序，你对指令的功能理解得越清楚正确，运用指令的能力就越强，在不同情况下能准确地选择合适的指令。写文章，描述相似事物类比的排比句，让人易读且更容易看清事物之间的联系，并有阅读美感；编程序，工控对象很多情况下也具有对称性，如果你编写的程序在逻辑关系上是对称性的，同样有阅读美感，并让人感到逻辑上值得信赖。写文章，有开头和结尾，概述式的开头和总结式的结尾前后呼应；编程序，有初始化的处理和结束的处理，数据的进入和数据的送出信息流向清楚。写文章，时不时地引用成语熟句，言简意赅，耐人寻味；编程序，按需求引用专门的指令，调用特殊宏汇集，梯级简单，处理隐藏。

在学习编程时，你不妨想像你在学习写文章，就像中小学时你的语文老师教给你的那样，先从造句开始，然后学习写段落，最后完成一篇文章，并在写作过程中积累词汇和学习表述，提高写作能力。我们的编程训练将循序渐进，从基本的指令开始，编写简单的梯级逻辑，然后学习围绕一个控制对象编写一段梯级逻辑的处理，最后完成一个小的项目，并在编程过程中，熟悉指令和训练思维，提高编程能力。

第 4 章

位逻辑指令编程

源自电气设备的时序逻辑控制，转为 PLC 的梯形图逻辑后，位指令当仁不让地占据了主导地位，可以说于梯级中无处不在，它们决定了梯级条件，或直接控制了输出设备。位指令是最简单的，也是最重要的，它们的运用是耐人寻味并值得探索的。

位指令：

- XIC 常闭继电器输入指令
- XIO 常开继电器输入指令
- OTE 非保持型继电器输出指令
- OTL 保持型锁存继电器输出指令
- OTU 保持型解锁继电器输出指令
- ONS 一次扫描有效输入指令
- OSR 前沿触发一次扫描有效输出指令
- OSF 后沿触发一次扫描有效输出指令

位指令操作分配的地址一定是布尔量的数据类型，用计算机的布尔量 0 或 1 表达了现场的开关量的开或关状态。

4.1 继电器输入/输出指令

在梯形图中，两条继电器输入指令和三条继电器输出指令几乎表达了生产工艺过程所有的顺序逻辑关系，它们或是来自现场的开关量传感器输入，或是传送至现场继电器设备的输出，或是梯级逻辑的中间结果。梯形图中大量的梯级都由基本的位指令的逻辑关系构成，如图 4-1 所示。位指令看似简单，运用起来颇有讲究，位指令所处的位置、输入指令和输出指令的搭配关系、输出指令的复位情况等都是值得研究和讨论的。

决定梯级条件的输入指令的逻辑位置安排，在梯级中并不是随意的，从程序执行的优化来讲，还是有规则可循的。例如 A、B、C 的 3 个输入点的位状态的波形如图 4-2 所示，根据梯级从左到右的扫描顺序，来判定输入梯级的位状态，一旦位状态无效，便不再判定后面的位状态，转到下一个梯级的扫描，所以要尽可能地将否定多的位编写在前面，以减少输入扫描的判断时间。每个梯级如此处理，积少成多，可节约不少扫描时间。注重每一个细节，不失为良好的编程习惯。

根据 A、B、C 的状态波形图分析，A 是条件成立最少的位状态；B 是条件成立次之少的位状态，C 是条件成立最为频繁的位状态，所以在梯级的输入条件应该编写梯级逻辑如图

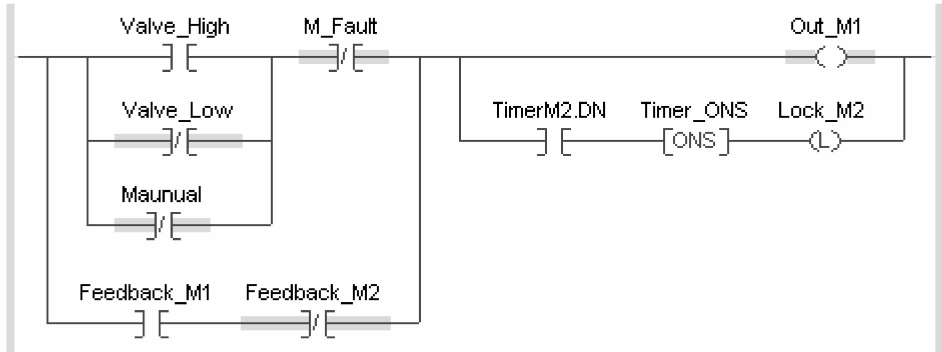


图 4-1　位逻辑实例

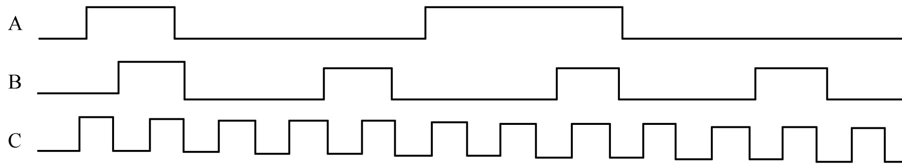


图 4-2　A、B、C 的 3 个输入点位状态的波形图

4-3 所示。

A 条件成立的机会最少，每当 A 条件不成立后，便不再判断后面的 B 和 C 的位状态，当 A 条件成立，B 一旦不成立，便停止判定 C 的条件，如此安排，可及早地停止判断，节省了扫描时间。

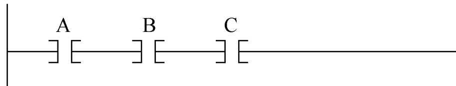


图 4-3　输入位逻辑顺序

留心一下有经验的工程师编写的程序，会发现多个输入位逻辑在同一个梯级中，前后的摆放位置不是随意的，不但考虑节约扫描时间的优化，并具有规律可循和一定的可读性，在指令互锁执行的情况下，甚至影响到指令的执行顺序。

非保持型输出指令 OTE 是一条最常用的指令，这条指令总是跟随梯级条件，当梯级条件成立，OTE 指令输出为 1；当梯级条件不成立，OTE 指令输出为 0。如图 4-4 所示。

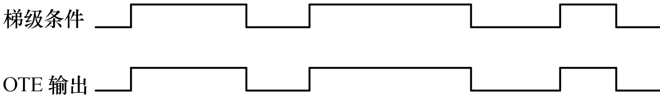


图 4-4　OTE 指令跟随梯级条件变化

如果你读到的梯级简单如图 4-5 所示或如图 4-6 所示，那么这是编程实现位对位的映像，使表示外部设备名称的内部标签与外部 I/O 数据一一关联起来。这种做法，跟别名标签的目的相似，如图 4-5 中的这些梯级逻辑可能被编辑在初始化例程中，对于输入数据来说，还可用来防止 I/O 数据更新与例程梯级逻辑扫描不同步而引起的问题，这是被推荐使用的方法。如图 4-6 的梯级逻辑则是为了将输出设备名称的标签与外部的输出模块的端点对应起来，这些输出设备名称就是输出端点回路所控制的设备，由例程运行而产生的控制结果。

特别要注意的是，当例程预扫描或后扫描时，所有的 OTE 指令将被置为 0。这是因为预扫描和后扫描将令所有的梯级条件不成立。其实我们应当关注每条指令对于预扫描或后扫描

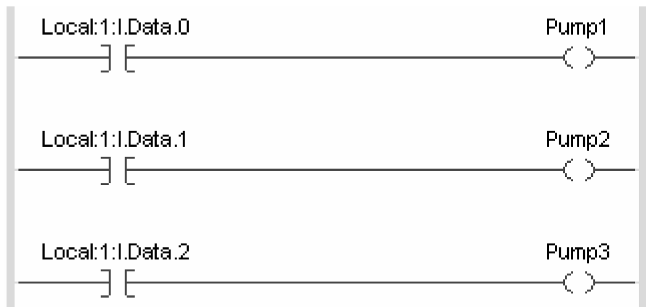


图 4-5　位映像梯级逻辑

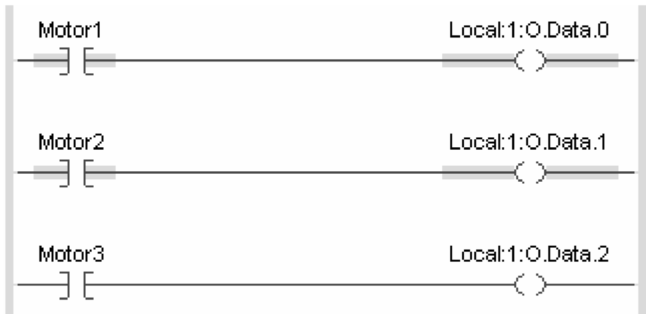


图 4-6　位映像梯级逻辑

的结果，但是 OTE 和 TON 指令总是被强调的，它们是最典型的非保持型指令。人们一般不大注意 OTE 指令，只因为预扫描之后，OTE 马上由梯级条件给出了它应有的状态，令人不易察觉而被忽略，而不是像 TON 指令那样，立刻可以看出 ACC 被复位为零了。

早期的编程常常可以看到如图 4-7 所示的梯级逻辑，这是硬件线圈通电后自保持回路的电路形式，根据这个思路，自然而然地编辑成了梯形图。锁存条件是 A，接通 B 后产生自保，直到解锁条件 C 作用。

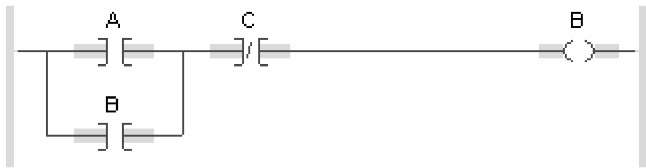


图 4-7　自保持梯级逻辑

这样的情形，也可以用一对锁存和解锁输出指令来编写，如图 4-8 所示。同样地，锁存条件是 A，作用后锁存 B；解锁条件是 C，作用后解锁 B。不同的是梯级结构，A 条件和 C 条件是相互独立和梯级对称的，表达了锁存和解锁的对称关系。

对比这两种做法，我更倾向于后一种，锁存关系就用锁存解锁指令来表达，直截了当，前一种做法带有技巧性，尽管这是众所周知的，没人读不懂。试想一下，如果我们想要了解控制系统中有哪些位具有锁存关系，前一种方式就很难统计出来，而后一种方式则只要用搜索功能搜索 OTL 指令，搜索结果便将所有的锁存关系列出来了。

细究图 4-7 和图 4-8 的梯级逻辑，会发现 A 和 C 的操作彼此并不独立，比如在 C 的解锁

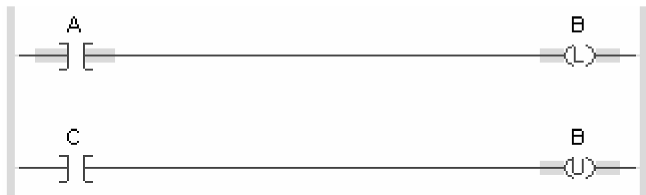


图 4-8　一对锁存和解锁输出指令

条件没有消失的时候，A 是没有办法再次锁存的。图 4-7 的 C 与 A 被串联在一个梯级，C 不解除，A 不起作用；图 4-8 的 C 条件位于后面一个梯级，即使 A 锁存了 B，下一个梯级的 C 马上解锁 B，最终送出的还是 B 被解锁的状态。

OTL 指令和 OTU 指令是可以解决这个问题的，使用锁存解锁指令，通常都要加上 ONS 指令，如图 4-9 所示，这样可以使得锁存和解锁两条梯级的执行分别独立。

可以在线测试一下，你会发现未加 ONS 指令的梯级，如果后级 C 的梯级条件没有解除，是无法给定 A 梯级条件锁存 B 的，因为 A 将 B 锁存，也就是 B 置 1 之后，紧接着扫描下面一个梯级，C 级条件的成立，马上解锁了 B，B 被置 0，直到例程所有的梯级扫描完毕，输出才能送走，所以最终得到的是后一个梯级的结果，即不能锁存的结果。

这样的编程，也许对外部的按钮会有限制，或是要求使用自行释放的开关，操作松开后能回到关闭状态；或是要求使用位置开关，锁存和解锁是选定的位置；或是操作手册要说明操作步骤，如锁存操作之前一定要关闭解锁等。

如图 4-9 所示的梯级逻辑编程方式完全可以避免后面梯级 C 条件的影响，因为 ONS 指令会令 C 梯级条件在一次扫描之后无效，所以不管 C 是什么样的状态，A 总是能将 B 锁存，这两个梯级的操作是相互独立的，不会彼此影响。这样位处理的编程，几乎对外部使用什么样的开关按钮没有限制，不管是瞬间的按钮还是固定的按钮，都可以通过编程完成对位地址的锁存或解锁。

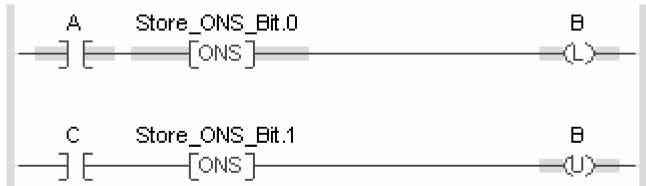


图 4-9　加上 ONS 指令的锁存解锁指令

4.2　单脉冲指令的用法

ONS 指令又称为一次性扫描有效指令，属于输入指令类型，系统将识别并放置在梯级的左边，ONS 指令一般不会单独使用，总是跟在梯级条件后面，将前面梯级条件的宽脉冲变为 ONS 指令后面梯级条件的单脉冲，以确保输出指令只执行一次。其波形如图 4-10 所示。

ONS 指令的编写，要求指定一个位地址作为 ONS 的存储位，用来记录本梯级条件已经一次扫描有效，自第一次执行完毕后，将存储位设置为 1，每次梯级扫描将检查此位，当检查存储位为 1，则令梯级条件不成立，不会执行后面的输出指令。存储位的复位，靠梯级条

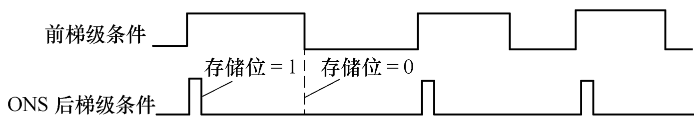


图 4-10　ONS 指令的波形

件的后沿执行，以准备下一次梯级条件的到来。

还有两条 ONS 之后拓展的单脉冲指令，这就是前沿触发指令 OSR 和后沿触发指令 OSF，跟 ONS 指令不一样的是，它们是输出指令。指令的输出位类似 ONS 产生的单脉冲，存在一个扫描周期，供后面的梯级逻辑使用，指令存储位的作用则跟 ONS 指令完全一样。

如图 4-11 或图 4-12 所示的编程，当宽脉冲的梯级条件使能 OSR 指令，在梯级条件的前沿触发一个输出脉冲，这个脉冲将用于某一个需要脉冲触发的输出指令的梯级条件，同时存储位 Push_Storage 置位，保存已触发状态，以确保只能执行一次，在 OSR 指令的梯级条件的后沿，即梯级条件消失时，存储位 Push_Storage 被复位，准备下一次的触发动作。

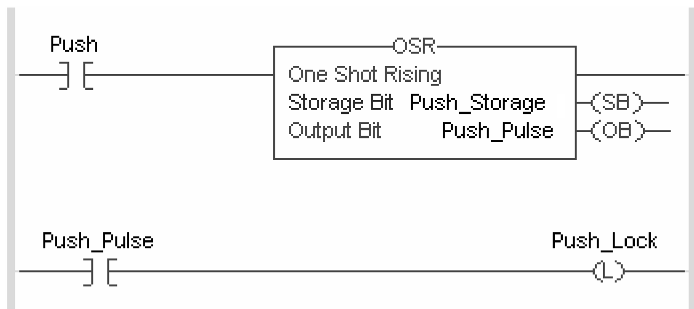


图 4-11　前沿触发脉冲

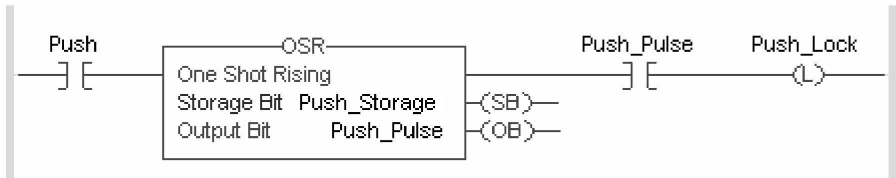


图 4-12　前沿触发脉冲

如图 4-13 或图 4-14 所示的编程，当宽脉冲梯级条件 Press 使能 OSF 指令，在梯级条件的后沿，触发一个输出脉冲，这个脉冲将用于某一个需要脉冲触发的输出指令的梯级条件，同时存储位 Press _Storage 置位，保存已触发状态，以确保只能执行一次。在 OSR 指令的梯级条件的前沿，即新的梯级条件的到来，存储位 Press _Storage 被复位，准备下一次的触发动作。

OSR 指令和 OSF 指令跟 ONS 指令有相似之处，但也有它们自己的特点，它们在宽脉冲的前沿和后沿触发，如果试图精确捕获宽脉冲的前沿触发时间点或后沿触发时间点，这两条指令就发挥作用了，这在现场实际中可能非常有用。

ONS 指令和 OSR/OSF 指令有什么不同之处呢？ONS 紧跟在一个梯级条件之后，专用来限制这个梯级条件的执行，比如说确保输出指令只执行一次。OSR 和 OSF 指令却能够产生

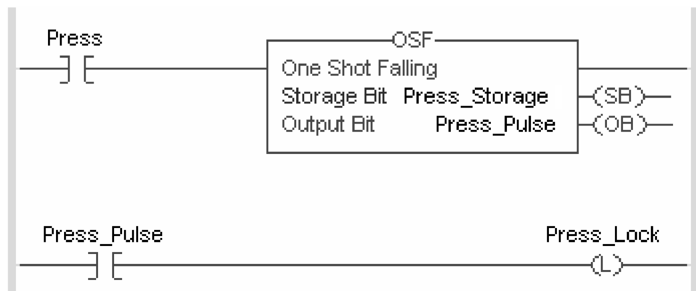


图 4-13　后沿触发脉冲

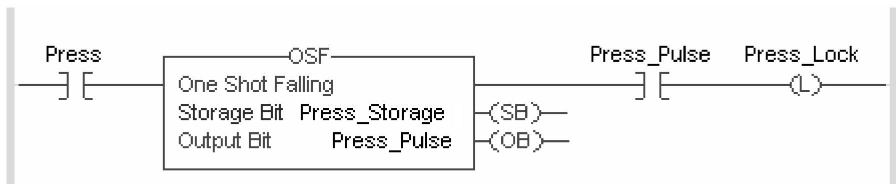


图 4-14　后沿触发脉冲

一个独立的输出单脉冲，这个输出单脉冲标签可以被多个梯级引用，在同一个例程中的梯级引用是可靠的，如果在例程之外引用，会有不确定性，因为这个脉冲只能保持一个扫描周期，例程之间的调用，特别是条件式调用，是难以保证在一个扫描周期内执行动作的同步。

一再强调的不可重复使用数据标签的提醒在这里同样重要，ONS 指令、OSR 指令和 OSF 指令的存储位的数据标签是不可重复使用的，否则会有不可预料的后果。存储位记录了对应指令执行曾经一次扫描有效的状态，只有梯级条件的后沿（ONS 和 OSR）和前沿（OSF）才能令它复位，共用数据标签将使得记录的状态混乱，从而引起执行动作的混乱。

尽管我们是如此地强调，ONS 指令必须紧跟在一个梯级条件后面，作为这个梯级条件执行的限制，如果有一天你看到有个例程的梯级逻辑编程如图 4-15 所示，就会感到迷惑了。这个梯级并不是不能执行，它会执行一次，这相当于梯级条件永远成立，但是它自己这个永远成立的梯级条件是没有后沿去复位存储位 Reset_ONS 的。

也许不远处的角落就藏着一个对它的解锁操作，如图 4-16 所示，在某个特定的条件下来执行这个复位，这样的编程就颇有点技巧了，只有对 ONS 指令的本质很清楚才能识破这点，我们可以通过搜索功能来找到这个隐藏着的复位。



图 4-15　无条件 ONS 指令



图 4-16　ONS 复位

如图 4-17 所示是一个牵强附会的编程，但这个实验让我们看清了 ONS 指令的实际作用，一般不建议你编写这样的梯级逻辑，我们还有更为直截了当的做法，除非你真的有一个奇怪的需求被迫如此。

还有，你不要指望预扫描会帮助一个无条件梯级的 ONS 指令的存储位复位，为防止误触发，规定 ONS 指令的预扫描是将存储位置位的。

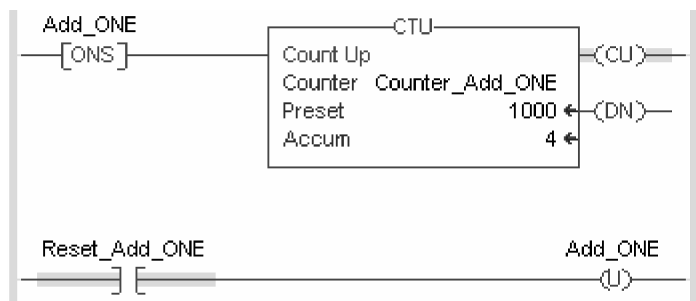


图 4-17　计数梯级逻辑

Push_Pulse			BOOL
Push_Storage			BOOL
Press_Pulse			BOOL
Press_Storage			BOOL

图 4-18　单独标签的内存消耗

考虑到内存使用的节约，ONS 指令的存储位又没有特殊的描述意义，通常会让多个 ONS 指令共同使用一个双整字，如 StoreONS. 0 是这个字的 0 位，用来做一条 ONS 指令的存储位，这样一个双整字就可以满足 32 条 ONS 指令的存储。稍加留心，会发现以上这个实例中 OSR 指令和 OSF 指令的标签是相当的耗内存的，作为单独标签的每个布尔量都消耗了一个数据基本单元，如图 4-18 所示。好好规划的话，我们可以让 OSR 指令和 OSF 指令存储位共用双整字标签，如同 ONS 指令做法一样，而每条指令的脉冲输出位完全可以是各自被控对象的用户自定义标签中的一个布尔量子元素。

4.3　位的互锁操作

这是一个经常遇到的问题，在现场逻辑控制中，需要对一些操作动作实施互锁来确保执行动作的可靠性。不要依赖于生产工艺过程的机械动作给予的状态来实施互锁，我们很难预计系统曾经停留在什么状态，残余的动作状态信息很可能给我们造成困惑，或者出现意想不到的状况，甚至应该互锁的动作发生了同时操作。

对于几个互锁执行的操作动作，采用锁存解锁指令对其控制是最有效和可靠的，有经验的工程师都会采取如图 4-19 所示的编程来确保互锁操作。

此例中有 4 个互锁的控制，每当满足其中之一的控制条件，便锁存自己的控制，解锁其他控制，不管其他控制当前的状态如何，这样可以确保只有一个控制在执行，这是一种十分可靠的做法，其明了清晰的表达，让读程序的人很容易理解，而不需要特别地去研究被控对象的当前状态和相关影响。

不要小看这个简单的方式，实践证明这是非常有效的。建议遇到动作互锁的操作时采用这个方式。

对于几个为数不多的互锁控制，我们可以采用位锁存解锁的方法来解决，如果有太多的互锁关系，也许采用指针和比较指令配合执行方式更合适，这将在后续的内容中讨论。

还有一点值得特别提及的，就是控制的惟一性。正在运行的同一个例程，或同时运行的几个例程中，千万不要让同一个输出点同时出现在几个不同的梯级，被不同梯级条件所控

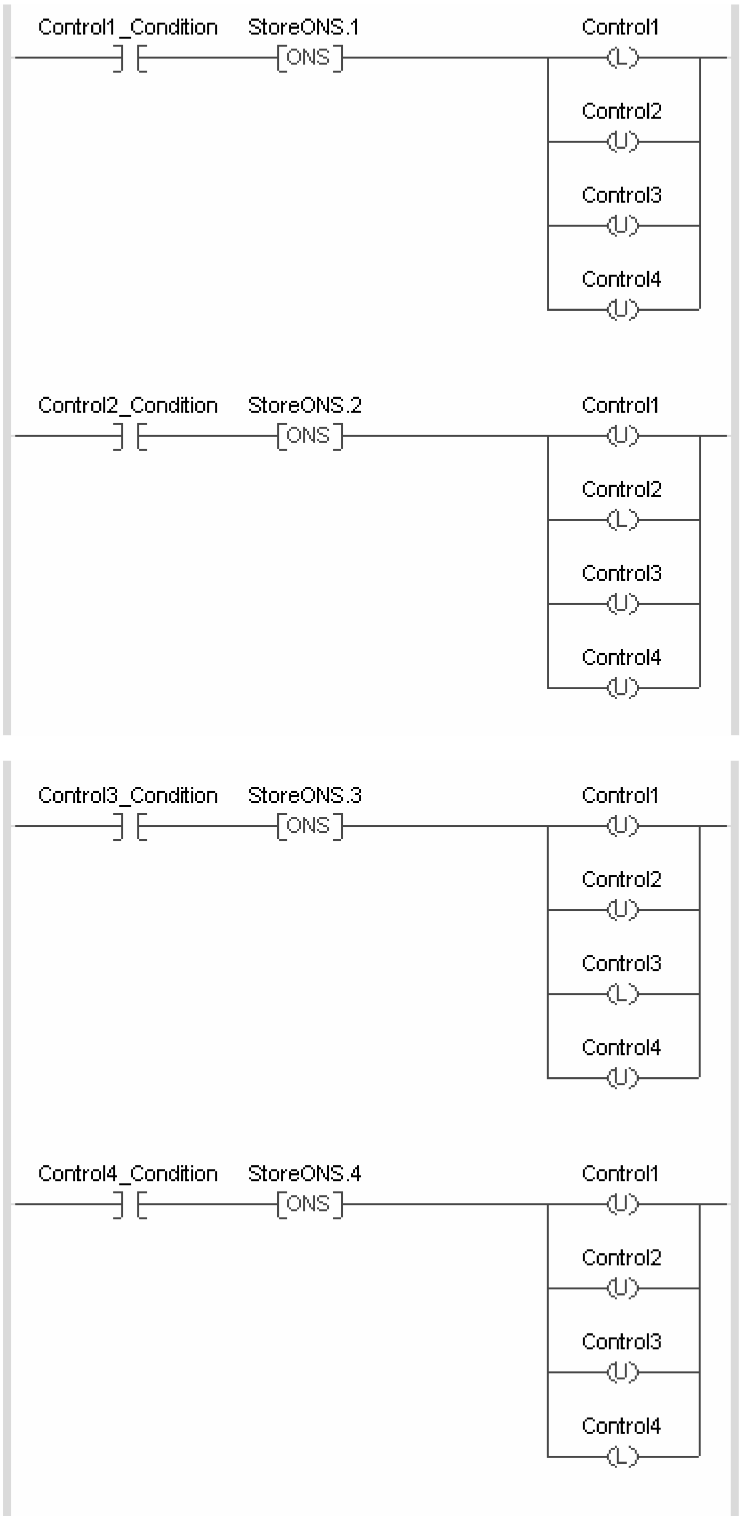

Control1_Condition
StoreONS.1
ONS
Control1
L
Control2
U
Control3
U
Control4
U
Control2_Condition
StoreONS.2
ONS
Control1
U
Control2
L
Control3
U
Control4
U
Control3_Condition
StoreONS.3
ONS
Control1
U
Control2
U
Control3
L
Control4
U
Control4_Condition
StoreONS.4
ONS
Control1
U
Control2
U
Control3
U
Control4
L

图 4-19　4 个互锁控制

制，这样会令前面的梯级控制统统无效。我常常看到我的学生面对着一个显而易见的梯级逻辑冥思苦想，因为没有看到他预期的逻辑结果，于是百思不得其解，其实就是后面的梯级正在修改他所期望的逻辑结果，那个曾经使用过又被他遗忘了的输出点的逻辑。如果编程一个输出被两个例程引用，那么这两个例程一定是不能同时运行的，比如说互锁执行的一个自动例程一个手动例程，它们共同控制着外部的一个开关。

当然这种情形多半出现在调试过程中，养成良好的编程习惯，可以减少错误，避免调试走弯路。其实，系统会提醒你这种不合适的行为，在例程或梯级校验时给出警告，这就是我们通常会忽略的警告信息。如图 4-20 所示是编程软件 Option 中的一个设置项，这个设置的含义是，使能在项目校验时探测到位地址的重复使用并发出警告，默认是选择。

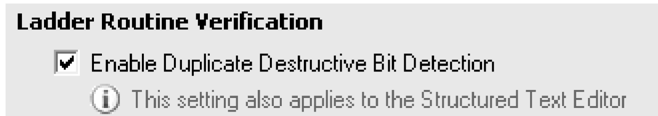


图 4-20　使能重复位探测

第 5 章

计时器指令和计数器指令编程

针对生产现场时序逻辑控制的需求而发展的工业自动控制系统，作为核心器件的控制器，无论其硬件是多么的简洁和廉价，其指令系统是多么的简约和低能，计时器指令和计数器指令都是不可或缺的。

尽管 Logix 控制器属于编程系统功能十分强大的工控产品，计时器指令和计数器的指令仍然是经常运用的基本指令。计时计数的操作来自 CPU 芯片基础功能的汇编语言，基于此开发的指令系统，不同的产品的运用方式是不一样的。作为指令系统非常成熟的产品，Logix 控制器的计时器指令和计数器指令应用是十分方便和简单的，清楚地了解指令的特征，正确地运用指令的状态位，尽可能地使用指令功能，将有利于编写的程序更为严密和准确。

计时器指令：

- TON　非保持型通延时计时器指令
- TOF　非保持型断延时计时器指令
- RTO　保持型计时器指令

计数器指令：

- CTU　增计数器指令
- CTD　减计数器指令

5.1　计时器指令编程

计时器指令是输出指令，根据梯级条件运行，计时基值为 1ms，计时范围为 1 ~ 2147483647ms，计时器计时数值为 32 位寄存器的正整数。计时器指令使用 TIMER 结构数据类型的标签，如图 5-1 所示。

⊟-Timer1	TIMER		Read/Write
⊞-Timer1.PRE	DINT	预置值	Read/Write
⊞-Timer1.ACC	DINT	累加值	Read/Write
Timer1.EN	BOOL	使能位	Read/Write
Timer1.TT	BOOL	计时位	Read/Write
Timer1.DN	BOOL	完成位	Read/Write

图 5-1　TIMER 结构数据类型标签

计时器结构数据标签的元素：

- PRE　预置值　设置预定的时间值，以毫秒为单位的数值，即计时脉冲个数。
- ACC　累加值　计时脉冲个数的积累数值，每当计时器指令被扫描时实施累加。
- EN　使能位　梯级条件满足计时器指令时，使能位置位。
- TT　计时位　计时器指令被使能，计时位置位，直到累加值大于等于预置值复位。
- DN　完成位　计时器指令被使能，完成位复位，当累加值大于等于预置值时置位。

我们先讨论非保持型通延时指令 TON。当作为输出指令的 TON 指令的梯级条件成立时，指令被使能，开始计时工作，直到累加值等于预置值，完成位置位，计时工作停止。梯级条件对于 TON 指令至关重要，当梯级条件消失，无论是否计时完成，累加值都会复位，同时所有状态位复位。通延时计时器工作状态位波形如图 5-2 所示。

TON 指令梯级条件消失引起的复位是常常被讨论的，这可以用来实现指令的自复位，从而得到定时操作功能，同时预扫描和后扫描对 TON 的复位影响在编程时也是需要关注的。

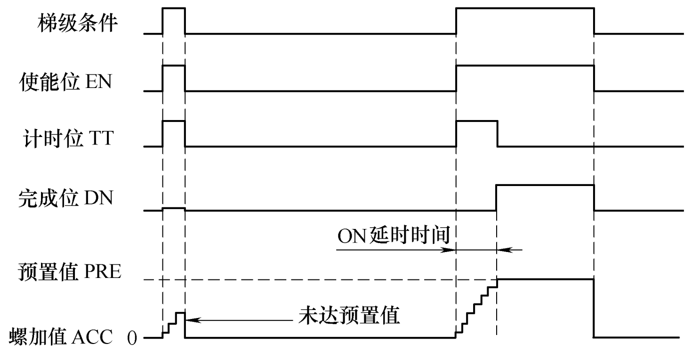


图 5-2　通延时计时器工作状态位波形图

非保持型的断延时指令 TOF，当作为输出指令的 TOF 指令的梯级条件不成立时，指令被使能，开始计时工作，直到累加值等于预置值，完成位从置位状态变为复位状态，计时工作停止。梯级条件转为成立时，无论是否计时完成，累加值都会复位，同时所有状态位回到初始状态。断延时计时器工作状态位波形如图 5-3 所示。

TOF 指令完全使用负逻辑工作，所以它的梯级条件、指令使能位和完成位与 TON 的同 3 个位是相反的，这条指令可以满足负逻辑关系的计时工作，但请留意它的计时位 TT 跟 TON 指令的 TT 位是一样的，或者说仍然是正逻辑关系。此外，它非常特殊的一点是指令初始状态并不是通常指令的复位状态，所以总是被强调，不要对它使用复位指令 RES，因为复位指令是将所有的状态位复位来作为初始状态。

保持型计时器指令 RTO。当作为输出指令的 RTO 指令的梯级条件成立时，指令被使能，开始计时工作，直到累加值等于预置值，完成位置位，计时工作停止。从计时对梯级条件的要求来看，RTO 指令与 TON 指令是完全一样的，惟一不同的是，当梯级条件消失，所有状态位复位，但累加值不会复位，累加值的复位与后面我们将要谈到的计数器指令一样，必须使用复位指令 RES。保持型计时器 RTO 工作状态位波形如图 5-4 所示。

PLC 从问世一直到现在都要用来控制时序逻辑、时序时序、时间的顺序，用计时器来担

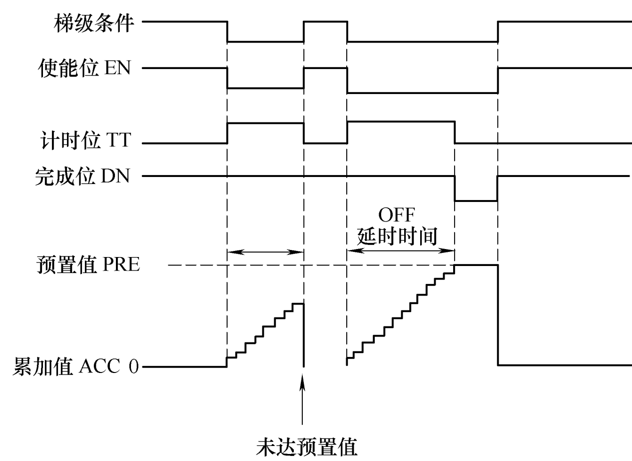


图 5-3　断延时计时器工作状态位波形图

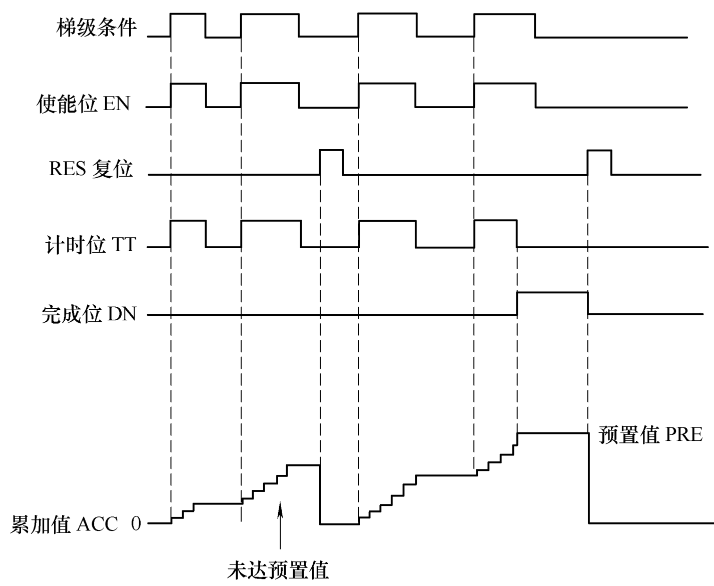


图 5-4　保持型计时器 RTO 工作状态位波形图

纲，当然是非它莫属了。计时器的工作通常有延时、定时和累计时。下面我们试着用一些常见的例子来编写计时器应用的梯级逻辑。

5.1.1　计时器的延时编程

延时动作指的是当事件发生后，并不马上执行，而是等到一段延时之后再执行，比如说，现场有一个传感器监视一个故障的发生，当探测故障发生的信号进来，如果马上动作，可能会引起停机，因为有的故障是需要停机的，假定这个故障信号并不是真正的故障，可能只是一个干扰的信号，停机就变成虚惊一场了，还可能带来不必要的停机损失。干扰信号或许是很小的一个窄脉冲，但在快速反应的 I/O 刷新和快速的例程扫描，已经足以使这个梯级

产生控制动作了，我们不妨让这个信号延时一段，确定故障真实的存在，再去故障停机。梯级逻辑编程如图 5-5 所示。

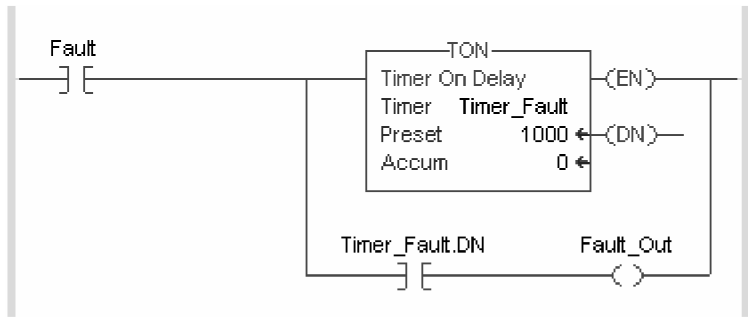


图 5-5　延时梯级逻辑

如果我们将计时器的预定值定义为 1000，作为时间基值为 1ms 的计时器，那么计时器 TON 的梯级条件 Fault 能保持 1s，这个故障输出动作的产生将延时 1s 执行。一个故障探测状态输入能够保持 1s 之久，我们有理由相信，故障真的发生了。如果这仅仅是一个扰动信号，不到 1s 便已经消失，计时器 TON 的梯级条件随之消失，计时器复位，完成位不会置位，故障输出动作不会发生。故障动作延时时间可以根据现场实际情况来确定，系统运行一段时间后，挑选一个合适的延时时间便可。

另一个延时监视的例子是对一个旋转机器的监测，以确定旋转的机器没有被机械卡死，一直处在旋转的状态。某个机器有一个机械触发器，如果这个机器一直在旋转，这个触发器会在设定的时间内触动一个感应器，感应器的状态将中断一个计时器的梯级条件，令计时器复位。如果在设定的时间一直没有中断，计时器的完成位便会置位，产生一个状态动作，给出机器不在旋转状态的信息。梯级逻辑编程如图 5-6 所示。

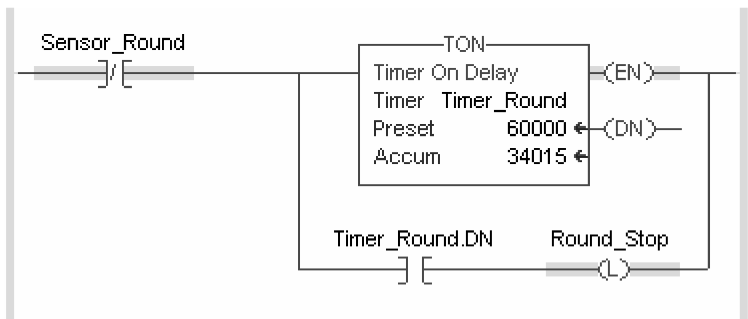


图 5-6　延时梯级逻辑

这个计时器的监视时间是 1min，如果 1min 过去，机械接触器还没有触碰到感应器，计时器的梯级条件在这一段时间都没有消失，计时器计时完成位将锁定机械旋转停止的信息。如果机器旋转正常，则感应器会不时地中断计时器的梯级条件，令计时器复位，完成位的动作就始终不能发出。

我们再看一个双向逻辑控制延时的例子，这是某个输出开关的控制需求，当控制发出打开的命令后，延时 2s 打开；或者控制发出关闭的命令后，延时 2s 关闭。如果发出打开的命

令后不到 2s 接受到关闭的命令，则不打开；如果发出关闭的命令后不到 2s 接受到打开的命令，则不关闭。请看如图 5-7 所示的梯级逻辑编程。

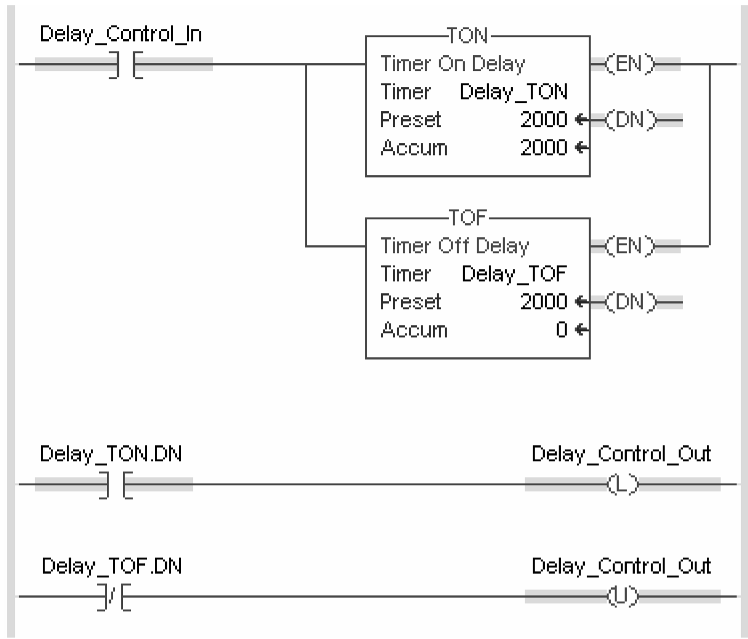


图 5-7　双向延时梯级逻辑

延时控制开关 Delay_Control_In 同时作为通延时器指令 TON 和断延时器指令 TOF 的梯级条件，开或关的任何情况下总能满足其中一个计时器的条件，并令它开始计时，最终产生它的延时动作输出；或者突然中断某个方向的延时而转回原来的方向。

这是一个典型的对称性的编程实例，巧妙地运用了通延时计时器和断延时计时器的逻辑动作，梯级简洁而明了。曾经有学生单用 TON 指令编出了我要的控制结果，由于没有使用断延时计时器，程序看上去没有对称性，被我无情地否决了。通延时计时和断延时计时的逻辑相反的特点，正是与我们的需求吻合的，为什么不这样直截了当地使用呢，没有对称编写出来的程序，逻辑关系是不够清晰的，解释颇费周折，过不久连自己也读不懂了，尽管可以实现逻辑功能却没有可读性，只要了解 TON 和 TOF 执行的不同之处，相信这几个梯级是可以令人一目了然的。

另外，要提醒注意的是，断延时计时器的 DN 是常开作为动作位（请回顾一下 TOF 指令的状态位的波形图）。梯级条件编写是当断延时完成动作时，也就是 OFF 时，去关闭延时控制输出。

5.1.2　计时器的定时编程

定时也是控制现场常见的需求，比如希望能够得到一个定时的动作脉冲去执行一个周期性的动作。

现在，我们来谈谈自复位计时器的定时作用，这是最常使用的方式，如果你看到一个梯级编成这个样子，如图 5-8 所示，你第一个反映就是，这将产生一个周期性动作，通常这个

计时器的 DN 位在后面的一个梯级中就被当做动作执行的梯级条件了，它所控制的输出指令不失时机地产生了一个执行动作。再强调一下，这个 DN 位一定要出现在这个计时器后面的梯级，即使你在自己的测试中，DN 位在前或在后似乎没有什么不妥，这样的执行顺序是一定不可以忽视的，同时也是一种良好的编程习惯，这可以帮助你避免某些陷阱。

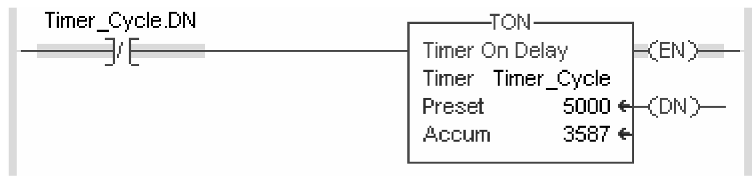


图 5-8　自复位计时

为了让大家更清楚定时动作的作用，我们不妨来分析一下这个自复位计时器的工作过程。自复位计时器永远都是用它自己的完成位常开作为它自己的梯级条件。我们知道，还没有开始工作的计时器，DN 位现在是为 0 的，作为常开节点，在梯级第一次被扫描时，梯级条件当然是成立的，此时计时器指令被启动，待这个梯级第二次被扫描时，梯级条件依然成立，计时器依然被使能，计时器指令继续执行，将上次扫描到这次扫描的时间间隔累加到 ACC 中，然后将 ACC 与 PRE 进行比较，如果小于，则离开梯级，这次指令执行完毕。如此循环多次，终于 ACC 几乎接近 PRE 了，在最后一次扫描中累加后的判断，ACC 大于等于 PRE，此时 DN 位置位为 1，指令执行完毕，离开梯级。等到这个梯级再次被扫描时，此时因为 DN = 1，毫无疑问，梯级条件不成立了，正如我们对 TON 这条指令执行的了解，当梯级条件不成立时，累加值复位，状态位复位。计时器指令用一个扫描周期的时间为自己复位，这就是自复位计时器。

这样，你当然明白为什么强调动作位一定要编辑在计时器指令的梯级之后了，如果你执意要编在不合适的位置，那你就把这个执行动作置于可能丢失的风险中了。

用两条计时器指令配合，产生两个时段的不同动作，且周而复始地进行，也是满足需求常见的编程方法之一。如图 5-9 所示的梯级逻辑编程，控制的结果将产生一个正负方向的方波信号，显然分别调节两个计时器的预置值，便可调节方波的频率和正负波占空比。传送指令 MOV 传送立即数至目标地址，将作为方波的幅值。

这也是自复位的计时方式，不过是两个计时器互为依存的复位，计时器 Timer_Reverse 的完成位的复位状态首先给 Timer_Positive 提供了梯级条件，当 Timer_Positive 计时完成，将为计时器 Timer_Reverse 提供梯级条件，直至 Timer_Reverse 计时完成，Timer_Reverse 完成位置位复位了计时器 Timer_Positive，复位后的 Timer_Positive 完成位立即复位了计时器 Timer_Reverse，至此完成一个周期。Timer_Positive 的完成位的不同状态则持续了两个时间段，用以完成正幅值和负幅值的传送。

再举一个耳熟能详的例子，交通红绿灯具有定时工作和互锁逻辑关系，比起工控系统中的多逻辑互锁，更为容易理解，我们就拿它来编程模拟工控系统中最常用的互锁逻辑动作。

假设东西方向绿灯和南北方向红灯亮 6s；东西方向红灯和南北方向绿灯亮 4s（为了观察方便，时间设得短一点），显然可以用两个自复位计时器来控制这两段时间，编辑如图 5-

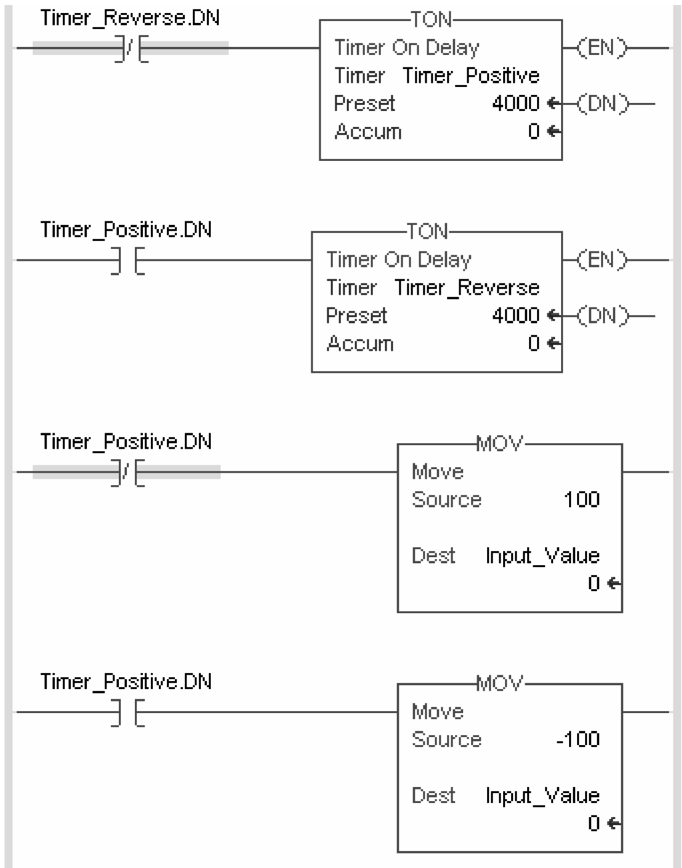


图 5-9　两个计时器互锁复位操作

10 所示的程序。

两个计时器各自复位自己的时间，然后将两个计时器计时动作互锁，每个计时器指令的梯级条件都是对方计时器的使能常开位，这是很常见的指令执行互锁模式（关于多指令执行互锁的编程方法，我们在后面再讨论）。

如何使用计时器的工作位来控制输出，我们可以采用两种不同的编程方式，这是输入梯级条件与输出指令搭配使用的一个典型例子。

如图 5-11 所示的方式，控制工作位是计时器的计时位 TT，即在计时器的整个计时时间段，TT 保持在置位状态，提供梯级条件支持非保持型的输出指令 OTE，点亮南北方向的绿灯和东西方向的红灯，或点亮东西方向的绿灯和南北方向的红灯，一旦梯级条件复位，相应的灯随之熄灭。

这是用持续的梯级条件来维持非保持性输出指令 OTE 的输出状态的一种编程实例。使用 OTE 指令时预扫描和后扫描的作用是要考虑的，显然在交通灯这样的例子中，影响是不太大的。

如图 5-12 所示的方式，控制工作位是计时器的完成位 DN，作为自复位的计时器，一般情况下完成位 DN 只存在一个扫描周期（互锁的计时器则不同），与之匹配的输出指令应该是锁存输出指令 OTL 和解锁输出指令 OTU，梯级编程为锁存南北方向的绿灯和东西方向的

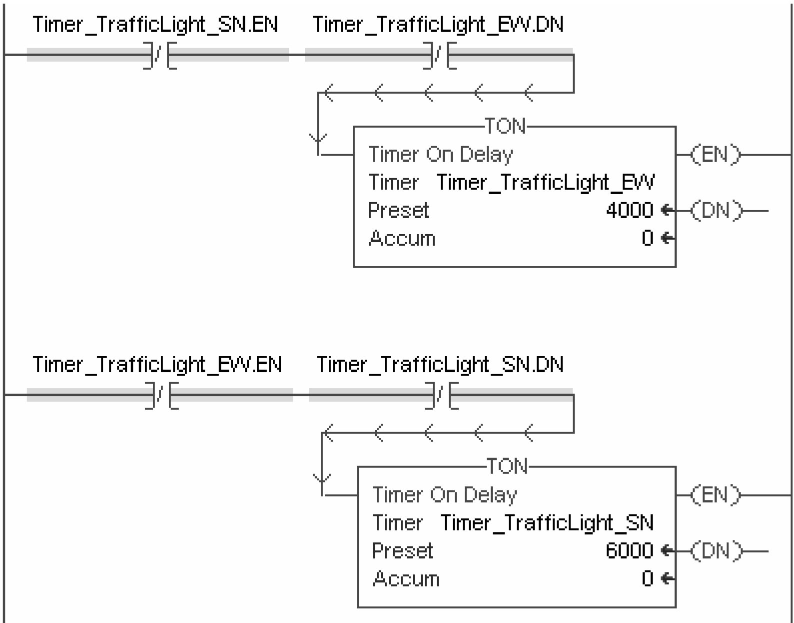


图 5-10　计时器互锁执行

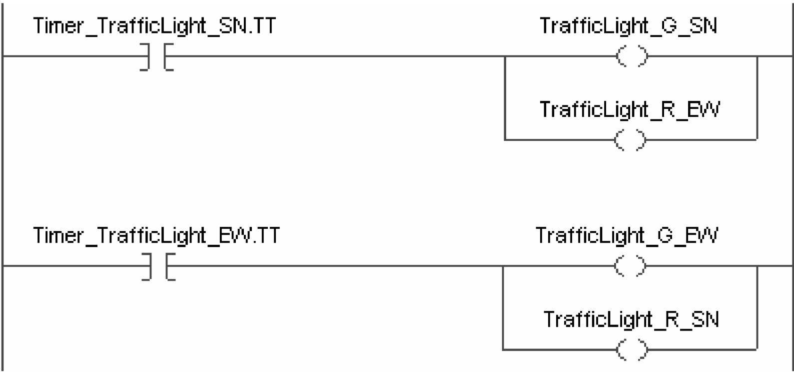


图 5-11　交通灯互锁控制

红灯，解锁东西方向的绿灯和南北方向的红灯；或锁存东西方向的绿灯和南北方向的红灯，解锁南北方向的绿灯和东西方向的红灯。

上面的两种编程方式，我个人更倾向于互锁计时器 DN 位搭配锁存解锁输出指令的方式，这样似乎更为严密而不受干扰。

让我们把这个程序下载到控制器中运行。任何自认为正确严密的程序，都要经过在线运行的测试，这就是现场调试。果然，问题暴露了，这个程序运行之后的前 6s 内没有任何灯被点亮。你几乎马上就想到了，这段时间没有任何一个计时器 DN 位置位，这些灯都是靠计时器的完成位 DN 位来动作的。应该一开始就让灯点亮，我们加上一个初始化的梯级，如图 5-13 所示。

原来，初始化也不一定是 0，只要是初始条件所需要的就可以了，凡是初始化的梯级或例程，它的执行条件一定是只有初次扫描执行惟一的一次。S: FS 被称为首次扫描状态位，

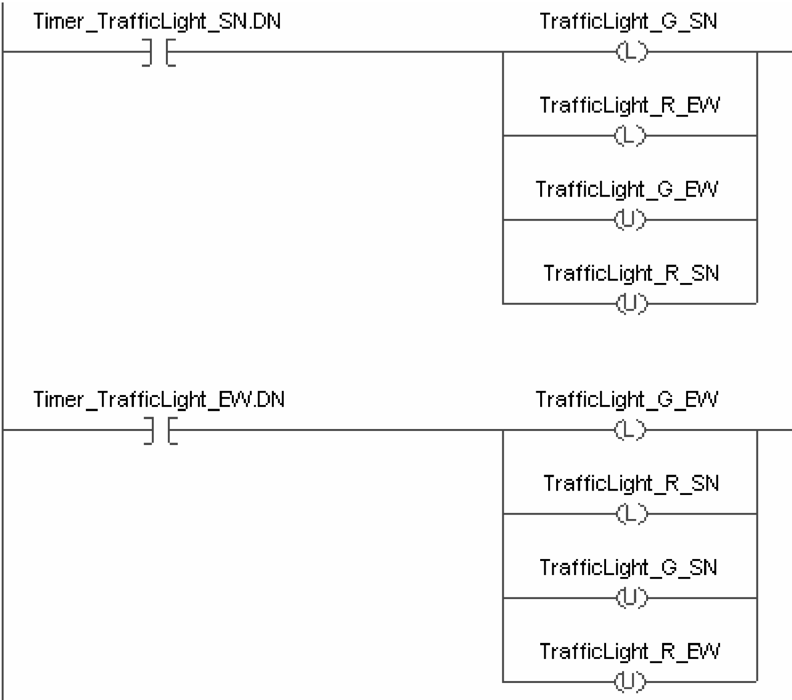


图 5-12　交通灯互锁控制

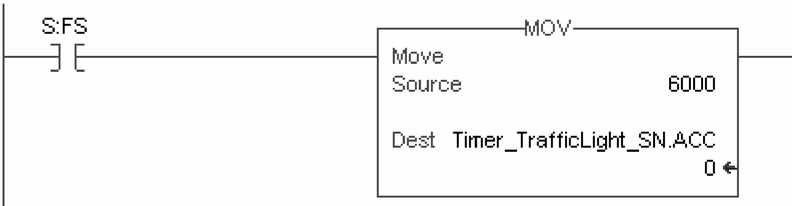


图 5-13　初始化梯级

是系统中少量的无需建立标签便可直接使用的关键字之一，这是仅仅存在一个扫描周期的为 1 的状态位，为初始化执行动作提供一个可使用的内部状态位。

有的编程人员会把项目中所有的初始化的操作集中在一个初始化例程里面，初始化例程中有多个如图 5-13 所示的梯级，然后用 S: FS 的梯级条件调用这个子例程。如图 5-14 所示，初始化例程 Initialize 只会在控制器运行之初执行一次。这是非常值得推荐的程序规划，尤其是我们谈到的标准化编程，就是希望把同类的处理集中于一处，这样便于查找，当我们猜测某个数据可能有初始化处理时，就会到初始化例程中去查找。

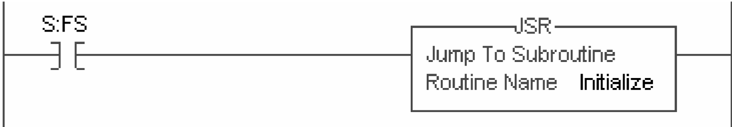


图 5-14　调用初始化例程

如果你不愿意将这个初始动作放置在远离梯级控制逻辑的地方，按照如图 5-15 所示的梯级逻辑编程也是可取的，这样将初始逻辑处理和正常运行的逻辑处理并列一处，让人一目了然，更有利于解读程序，这也是早期有经验的工程师经常采取的方法。从这个实例看来，标准化的编程似乎在挑战这个典型的模式，事情往往这样，标准化的东西总有些繁杂，但有规矩可循，简洁的手段就不愿意墨守成规。如果是一个团队的工作，恐怕还是标准化更有意义，否则沟通所花费的时间将更多。

这个初始处理并列一个初始化条件，输出指令则利用了一个常规的完成操作，似乎比 MOV 一个值更为简洁，但不适合单独梯级处理。

图 5-15　含初始化的梯级逻辑

现在，这个交通灯控制的梯形图的完整梯级逻辑如图 5-16 所示，请注意梯级安排的顺序。计时器在前面的梯级，它的完成位所执行的动作梯级在后面的梯级。控制南北方向和东西方向的两个互锁的计时器，按照梯级如此的排列，首先计时的是东西方向的计时器 Timer_TrafficLight_EW，所以 S: FS 锁定的是南北计时器的动作，令南北方向率先流通，一旦东西方向计时器完成，便进行交换，之后循环下去。

还有一点需要我们记住的就是，计时器的动作位只有 DN 位和 TT 位，这是指令计时开始之后具有的两种状态，只有它们跟计时过程有关。有人居然采用 EN 作后面梯级逻辑的动作位，那个程序运行起来看上去也还正常，要知道那恰恰是一对互锁的计时器，用于限制两条计时器指令的同时执行，这使他浑然不觉自己的错误。计时器的 EN 位永远跟梯级条件一致，要知道是梯级条件令计时器使能，如果你使用计时器的 EN 位，那你还不如直接用计时器的梯级条件，那还要计时器的介入何用呢。

不过，看上去不规范的运用，不是愚昧，就是智慧。我曾读过一段例程，看到 EN 突兀地出现在一个梯级，吃了一惊，细读之后，不禁哑然失笑。原来这个例程里有一个堪称主心骨的计时器，它的梯级条件十分复杂，并行了好多分支的位状态，一个屏幕都看不完。这个计时器之后一百多条梯级后，居然有个报警动作，也要使用这个跟计时器相同的梯级条件。我刚才还在说，不如直接用这个梯级条件，还真有这种做法，真是开眼界了。

话又说回来，如果出于愚昧，使用了 EN 位，那就不可原谅了。

有时候程序运行的陷阱是我们自己设置的，因为我们没有正确地使用指令，没有正确地使用指令的参数。

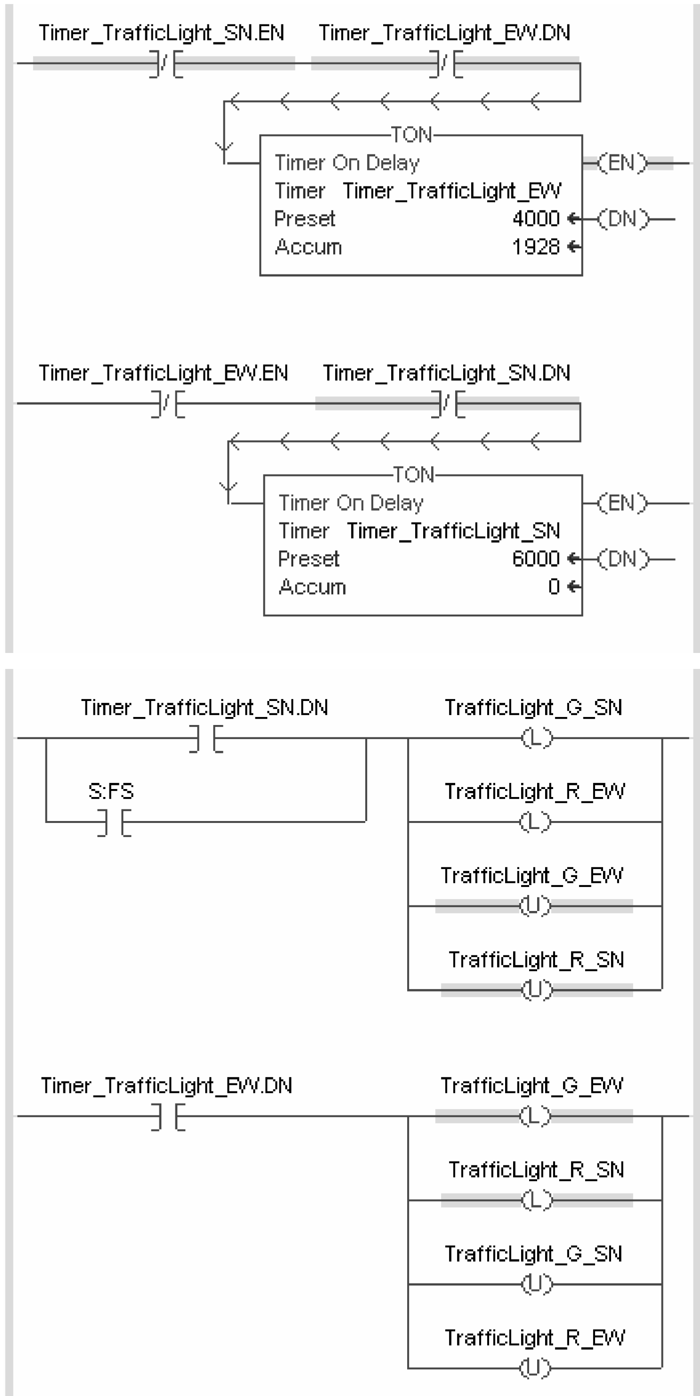


图 5-16 交通灯控制的完整梯级逻辑

5.1.3 计时器的累计时编程

计时器还有一个重要的用途，就是用于时间的累计。如果计时器的梯级条件是断断续续的情形，只要梯级条件成立，就开始时间累计；梯级条件不成立，就停止时间累计，最好的

选择是采用保持型的计时器指令 RTO 来完成。如图 5-17 所示的梯级逻辑编程，就是累计时的一种编写方式。

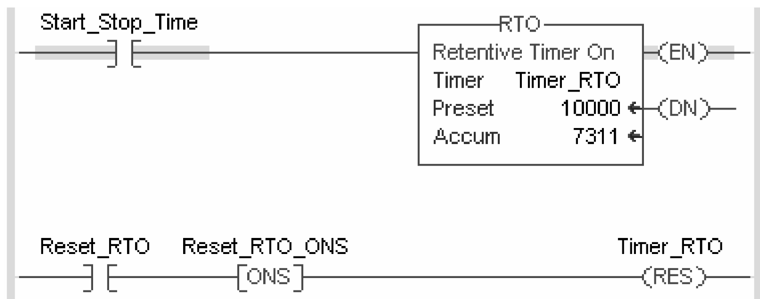


图 5-17　累计时梯级逻辑

保持型计时器累加值使用梯级条件对计时器复位，复位的梯级条件可用 ONS 指令加以限制，以确保只会执行一次，否则在复位的梯级条件没有取消时，计时器不能开始计时，一般情形，复位指令最好确定只能执行一次。

这里有一个典型的时间累计的例子，要求编程采集一个发动机运行的累计时间为机组保养提供依据，当发动机的转速超过 1100 转/分钟，算作发动机的正常运行时间，将这个时间进行累计；当发动机转速不到 1100 转/分钟，发动机不算正常运行时间，不做时间积累，这是一个允许梯级条件断断续续的计时累积，比较指令的判断结果作为梯级条件，决定是否进行计时累积，梯级逻辑编写如图 5-18 所示。

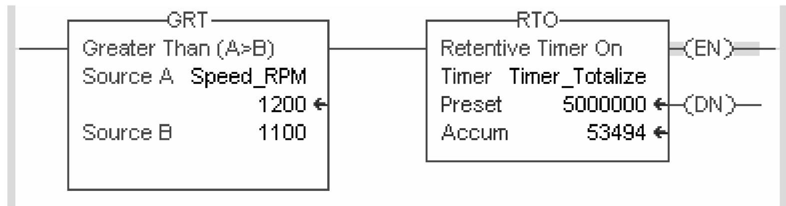


图 5-18　发动机累计时梯级逻辑

当时间累积达到设定的预定值，计时器的完成位 Timer-Totalize DN 置位，给出机组保养有关的提示。

是否只有断断续续的时间累计才会用到保持型计时器 RTO 指令呢？也不尽然，出于一些特殊的目的或需求，有时也会用到 RTO 指令。

在现场运行程序实例中，摘过来两个这样的梯级，如图 5-19 所示。这是一个电动机连续运行和停止的时间累计，运行或停止的梯级条件满足，连续时间达到 1h，计时器的完成位 DN 将执行相应的动作；不足 1h 梯级条件消失，指令未使能，计时器常开使能位令计时器复位。

乍看来，这是利用计时器梯级条件不满足，用计时器未使能状态来给计时器复位，为什么编程者不采用 TON 指令来完成呢？如前面的讨论，我们已经知道，当 TON 指令的梯级条件消失的时候，计时器的累加值是复位的。在这个实例中，满足的需求显然是：除非梯级逻辑自己判断的不足 1h 的运行或停止的梯级条件消失可以复位外，其他外在的原因引起的计时器累加值复位都是不允许的。

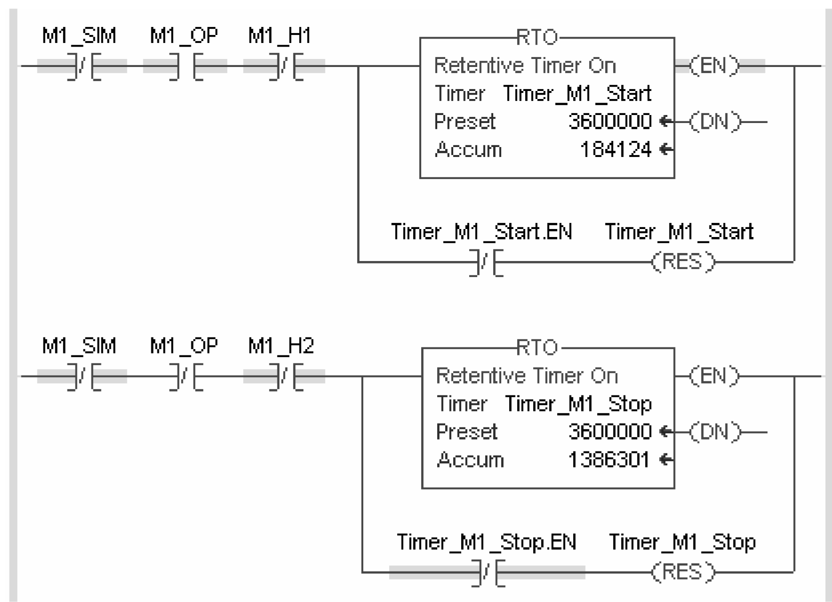


图 5-19　连续累计时梯级逻辑编程

这里重要的差别在于，RTO 指令的累加值不受预扫描和后扫描的影响，它的累加值在经历了预扫描和后扫描之后，仍然能够继续积累而不会被清零。你可以试着将控制器从编程状态切换到运行状态，反复进行几次，可以观察到控制器在编程状态时会停止累计时间，在运行状态会继续累计时间，累计值在运行和编程反复切换的过程中不会被清除。如果是 TON 指令的话，每次从控制器运行切换到编程状态，它的累加值会静止在当前的数值，一旦控制器从编程切换到运行，程序的预扫描作用就会将累加值清除为零，然后重新开始计时。只有在线调试时，才知道自己什么地方考虑不周，如果对指令有足够详尽地理解，就会迅速反应过来，知道自己缺失在何处。

从保持型计时器 RTO 运行的状态波形图中，我们早已了解它的累加值是不能利用梯级条件的消失来复位的，保持型计时器和保持型指令计数器一样，必须要用专用的复位指令 RES 来复位。

说到复位指令 RES，有一件事不能不再次提及，这就是对断延时指令 TOF，千万不可用复位指令。不知是出于习惯还是什么原因，通常我们说的置位，就是把这个位地址设为 1；而我们说的复位，就是把这个地址设为 0。当使用复位指令 RES 时，它的复位操作是令所有位为 0。但是 TOF 的初始状态可不是 0，大家已经看过它工作的波形图了。可以说除了 TOF 指令，其余指令的复位状态都是 0，只有它是个特例。在编程满足需求时，如果你想给它复位的话，设法消除它的梯级条件就可以了。前面在介绍 TOF 指令时，已经提到不能使用 RES 复位，但是想来不会被注意，往往在面对真实的应用时，人们才会恍然大悟。

从以上编程的一些实例来看，大家可以明白，选择什么样的计时器指令来满足你的需求是大有讲究的，计时器指令看起来简单，运用起来却是非常灵活的，许多例程的编写，只有通过现场调试，不断地改进，才能找到最佳的解决方案。

每种控制器产品都有计时器，但是它们的用法未必相同，如果你使用的是 Logix 控制

器，那么享受功能齐全的计时器给你带来编程乐趣吧。

5.2　计数器指令编程

计数器指令是保持型的输出指令，当梯级条件跳变时实施计数操作，计数范围为 -2147483648 ~ 2147483647，这正是 32 位寄存器的 DINT 数据类型可表达的正负数据范围，计数器指令使用 COUNTER 结构数据类型的标签，如图 5-20 所示。

⊟-Counter1	COUNTER		Read/Write
⊞-Counter1.PRE	DINT	预置值	Read/Write
⊞-Counter1.ACC	DINT	累加值	Read/Write
Counter1.CU	BOOL	增计数使能	Read/Write
Counter1.CD	BOOL	减计数使能	Read/Write
Counter1.DN	BOOL	完成位	Read/Write
Counter1.OV	BOOL	上溢出位	Read/Write
Counter1.UN	BOOL	下溢出位	Read/Write

图 5-20　COUNTER 结构数据类型标签

计数器结构数据标签的元素：

- PRE 预置值　设定计数预定值；
- ACC 累加值　按照梯级条件跳变进行的计数积累；
- CU 增计数使能　增计数指令梯级条件跳变置位；
- CD 减计数使能　减计数指令梯级条件跳变置位；
- DN 完成位　当累加值大于等于预置值时置位；
- OV 上溢出位　当累加值达上限值 2147483647 时计数加 1 后，该位被置位，且回归到计数下限值 -2147483648，此时完成位仍置位；
- UN 下溢出位　当累加值小下限值 -2147483648 时计数减 1 后，该位被置位，且回归到计数上限值 2147483647，此时完成位仍复位。

增计数指令 CTU 完成加 1 的操作，如图 5-21 所示。增计数指令的梯级条件跳变，位地址 UP_Pluse 从 0 变为 1 时，增计数使能位 CU 置位，累加值加 1，梯级条件消失；位地址 UP_Pluse 从 1 变为 0 时，增计数使能位 CU 复位，等待下一次的计数。显然，是 CU 位的置位引起了增计数。每次增计数完成，计数器都要测试，比较累加值 ACC 和预置值 PRE，以决定完成位和上溢出位的状态。

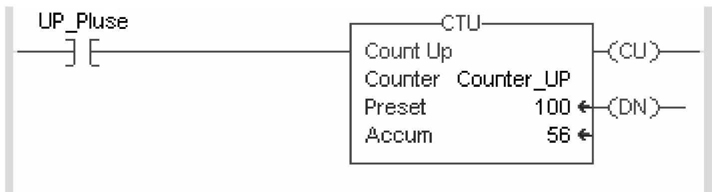


图 5-21　增计数指令完成加 1 的操作

减计数器指令 CTD 完成减 1 的操作，如图 5-22 所示。减计数指令的梯级条件跳变，位地址 Down_Pluse 从 0 变为 1 时，减计数使能位 CD 置位，累加值减 1，梯级条件消失；位地址 Down_Pluse 从 1 变为 0 时，减计数使能位 CD 复位，等待下一次的计数。显然，是 CD 位的置位引起了减计数。每次减计数完成，计数器都要测试，比较累加值 ACC 和预置值 PRE，以决定完成位和下溢出位的状态。

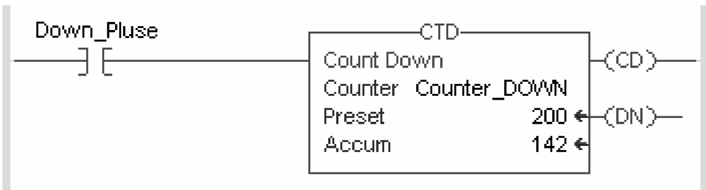


图 5-22　减计数器指令完成减 1 的操作

计数器指令在控制器中的作用跟计时器指令一样，都是最基本的功能，但是它的执行却完全不同，这主要表现在对梯级条件的要求不同，计数器指令的执行，全靠梯级条件跳变来触发，不管是增计数还是减计数。计数器计不计数只跟梯级条件跳变有关，跟 DN 位置位不置位无关，完成位在这里只是一个标志位，表示当前累加值大于等于预置值；跟上溢出位置位不置位无关，那只是累加值 ACC 的进位状态；跟下溢出位置位不置位无关，那只是累加值 ACC 的借位状态。总而言之，只要梯级条件跳变，计数器指令必定计数，然后经比较判断后给出状态位。

计数器是否计数不受状态位的影响，而计时器是否计时跟状态位紧密相关。

计数器有时用来记录产品的产量，如图 5-23 所示是一个产品数量计数器，每当产品通过时触发感应器，感应器状态位 Touch_Bit 作为计数器的梯级条件发生跳变，计数器就计数一次。当计数需要重新开始，可由操作员的外部操作来清零计数器，只要在人机界面设置针对位标签 Reset_Counter_Products 按钮操作便可得到复位的梯级条件。同时，计数器的累加值也可以送到人机界面显示。

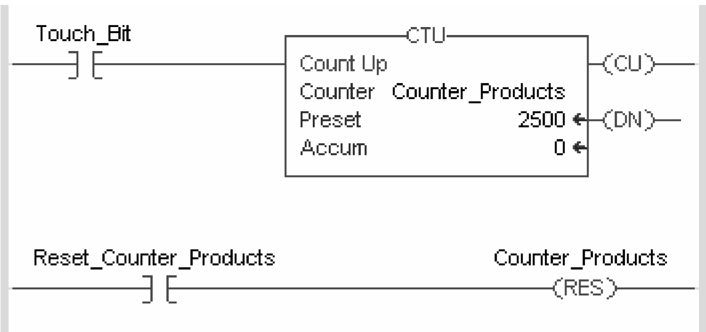


图 5-23　产品数量计数器

按照计数器指令的工作过程，通常的情况下，计数器指令前面必定是要一个梯级条件来作为计数的脉冲，可是一个例程中却看到了一个没有梯级条件的计数器指令，甚至连预置值都没有设置，如果这个例程正在运行，你会看到计数器在飞快地在计数！这真令人不解！如图 5-24 所示。

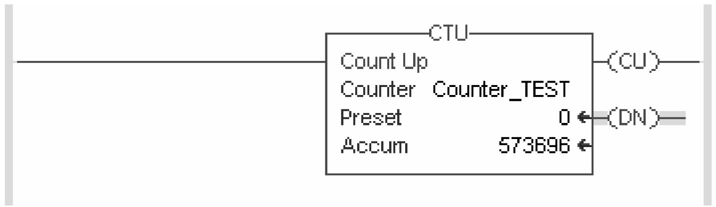


图 5-24　无计数脉冲触发的计数

这时，你不妨使用一下编程软件的搜索功能，可能在离这个梯级的很远之处，或者根本就在例程末端处，你会看到一个这样的梯级，如图 5-25 所示，原来躲在暗处一直给计数器指令复位的就是它。这回，你总算看透了计数器，只要它的状态位使能被触发就会计数。原来，提供一个正向脉冲梯级条件使能计数器和复位计数器的常使能位，对计数器产生的作用都是一样的。这个例子编程实现的显然是在计算例程被扫描的次数，因为这样的复位方式让计数器指令所在梯级每逢扫描就会加 1，无条件的梯级是每次扫描都会执行的。

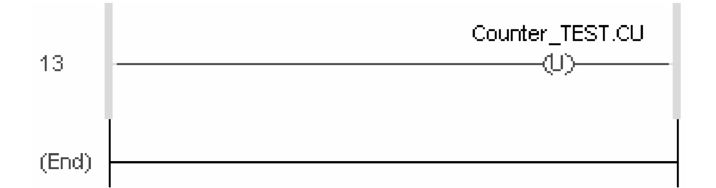


图 5-25　计数器使能复位

计数器不尽然都是用来计数的，也可以利用它不断变化的累加值来达到某种目的。例如某个自动控制系统中，有一个主控制器和多个从控制器，为了保证它们的联动准确无误，必须让每个从控制器时刻了解主控制器的工作状态，确定主控制器正在运行中。因此，主控制器需要发出一个连续变化的数据作为脉搏信号让从控制器读知。不妨采用一个计数器来产生这个连续变化的数据，就用刚才的方法，编写梯级逻辑如图 5-26 所示，计数器的预置值没有确定的意义，仅仅是为了让数据连续变化的重复，即使没有设定预置值也不会影响计数器的连续计数，到一定的时候，累加值自然会溢出翻转。

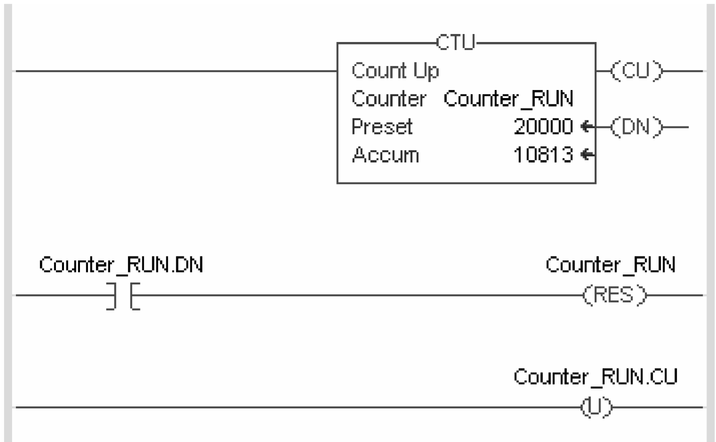


图 5-26　产生连续变化的数据的梯级逻辑

计数器指令所在的梯级每次被扫描时，会完成加 1 的操作，如果设定预置值，并令完成位 DN 位置位时复位计数器，此时运用趋势曲线 Trend 来观察计数器的累加值 Counter_RUN. ACC，可以观察到一个锯齿波的图形，如图 5-27 所示。关于趋势曲线显示组态的详细介绍，可参考《ControlLogix 系统实用手册》一书。

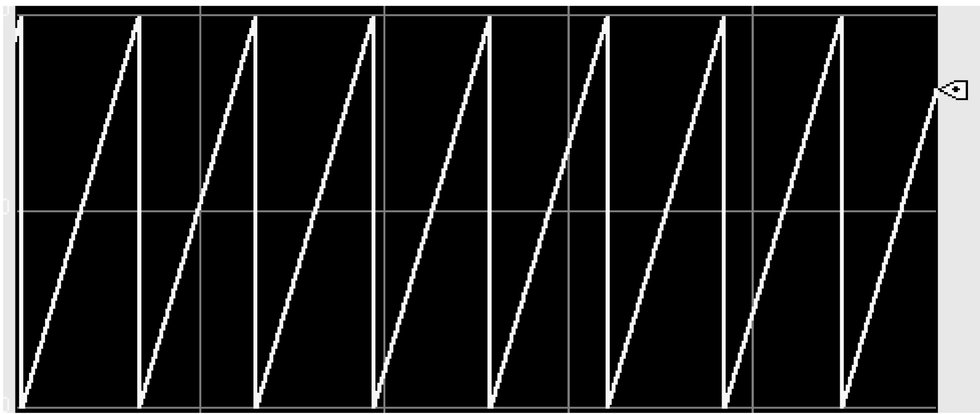


图 5-27　一个锯齿波的图形

如果觉得每次扫描计数累积太快，不妨用一个位分频来产生计数脉冲，分频的梯级逻辑编写如图 5-28 所示。注意看第一个梯级，自动地实现了 0 和 1 的翻转，在早期的程序中这种方式常被使用，也许是深受硬件方式的影响，这分明就是触发器的翻转动作，也许是当时太注重节约内存，或者是特别喜欢用位来完成一些技巧性的编程。观察这个位指令变化的梯级，总会折服于它的简单和精巧，显然每两次扫描将会产生一个计数脉冲。如果想要再一次分频，只须对这个地址再增添同样的一个梯级，重复一次这样的操作，计数器使用最后一个结果的变化位即可。

分频产生的计数脉冲，给计数器提供了触发条件，连为计数器复位的梯级也可免去。

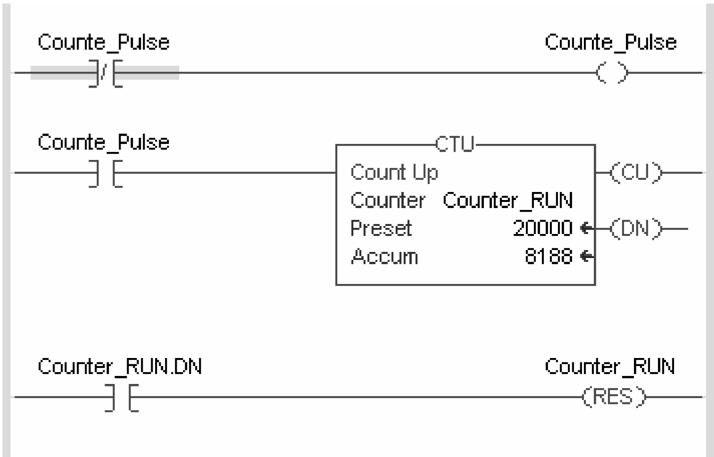


图 5-28　分频的计数

同样采用趋势曲线监视这个计数脉冲分频后的计数器累加值数据，可以看到更为稀疏的锯齿波形，频率正好是原来的一半，如图 5-29 所示。

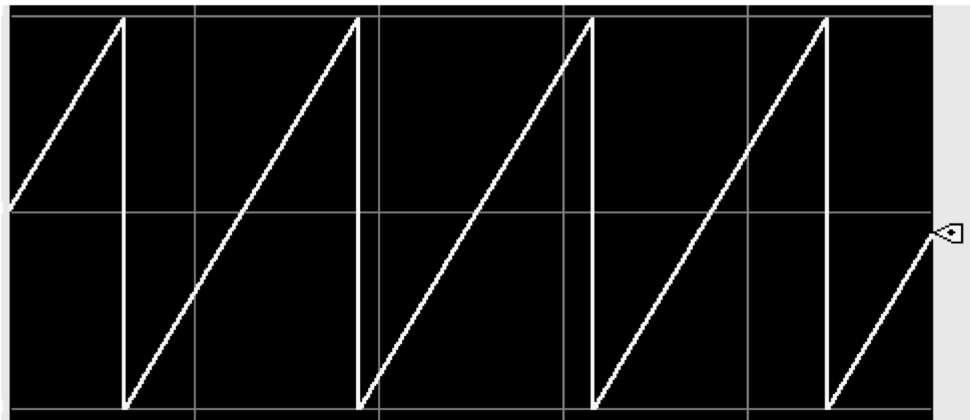


图 5-29　分频后的锯齿波形

如果我们想要得到一个可逆的双向计数器，编写如图 5-30 所示的梯级逻辑即可，虽然我们一直强调，不可重复使用标签地址，否则将出现不可预料的结果，但是在这个例子中你可以看到，我们正是利用增计数指令和减计数指令两个不同的梯级条件跳变，共同修改同一个计数器结构数据标签，才实现了可逆双向计数器的功能。无独有偶，类似的例子在后面的章节中堆栈数组处理还能再次见到。

增计数和减计数是不同的梯级条件，更不能同时跳变，否则计数器的累加数将看不到变化。

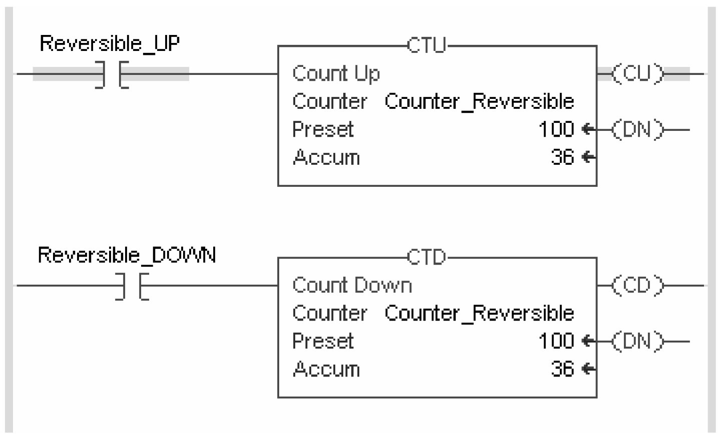


图 5-30　可逆计数的梯级逻辑

再看一个比较特别的实例，这是满足人机操作界面的一个下层处理，设定固定的设置值。在控制器中编写梯级逻辑如图 5-31 所示。

当人机界面 PB 按下按钮时，计时器开始计时，计时器的预置值时间作为操作的滞留时间，计时器完成位提供了一个计数脉冲，计数器加 1。对于外部来说，按住设置按钮，设定一个预期的数值，这个数值设置成功后被反馈到人机界面 PB 的屏幕上，让操作员看到被设

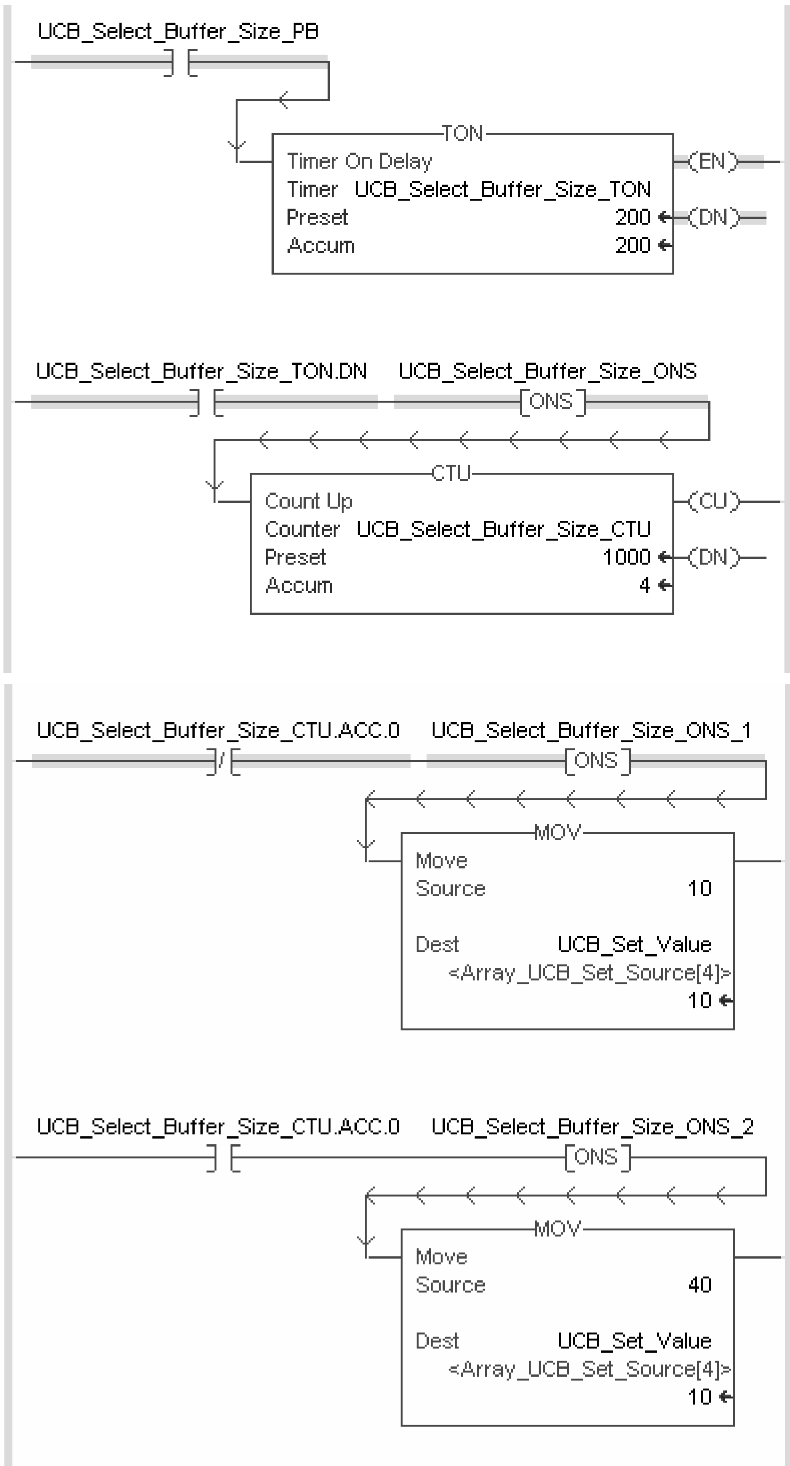


图 5-31 设定固定设置值的梯级逻辑

定值后松开设置按钮。此处计数器的预置值没有使用意义。

后面梯级的两条 MOV 指令分别传送了两个不同的设置值，它们的梯级条件是计数器的

ACC 值的最低位，作为不断计数的累加值也遵循二进制的计数规则，每增加 1，此位要翻转一次，一次为 0，一次为 1，轮番交替。ACC 值最低位的 0 和 1 的状态为两条 MOV 指令提供了不同的梯级条件，输入位指令一个是常开另一个是常闭。

这样区分了两种设置值的设定情况，对于操作员来说，他所感受的是每当他按下按钮时，就会显示出他选定的设置值，只有两个数据供他选择，他可以翻转着寻找。0.2s 的滞留时间对于操作员来说，只是按了一下按钮，即使按住更长的时间，也不会影响操作的结果。这种做法是可以类推到多个选择的，至少我们知道，原来我们使用的选择参数界面，底层有着如此的处理。

这段摘抄的梯级逻辑，初看时我也感到有点疑惑，既然是人机界面的按钮操作引起计数器计数，然后用计数器的 ACC 值去区分两种操作，为何不直截了当地将按钮作为计数器梯级的条件呢？细比之下，惟一的差别在于，这样的做法在操作上是有点延时的，也许在这个实例中，需要一点儿延时，于是采用了这个方式。也许这是出于保护的目的，这样人机界面偶尔的一次触碰是不会改变数字设置的，只有操作员有意地停留延时时间，才有数据重设的结果。正如我们操作手册上写着的，按住按钮 1s，更换设置数据，诸如此类。

从另外一个方面来看，这是一个通信极为节约的方式，只要一个按钮的通信，就解决了多个设置，如果每个按钮对应一个设定值，就需要多个通信连接了。早期由于资源的紧缺，在节约上真是不遗余力地施展技巧。也许现在的编程无须这样追求节约资源，改为直截了当的通信，让更多的人更容易明白编程者的意图。抑或这根本就是一个早期的人机界面操作的处理模式，一个流传至今的巧妙的翻转选择数据的方式。

不管怎么样，作为学习者愿意看到更多样式的梯级逻辑执行形式，这对我们解读前辈们编写的程序大有好处。如今前辈们编写的程序还大量地运行在系统中，作为系统维护人员的新手们需要了解各种各样的思维模式。我们不采用这种编程方式并不等于我们不要了解这种编程方式。

有时看上去笨拙的处理，却是现场调试的结果，有些动作处理在逻辑上是很简单的，但在现场实施却因硬件或机械的限制，或者现场信号的原因，我们不得不改用妥协的处理方式。

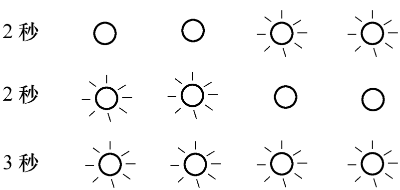


图 5-32　亮灯的变化过程

这里再例举一个比较原始的编程方式，了解一下过去是多么地喜欢利用位来操作。这是利用计数器的累加值 ACC 的二进制数的特征来编写的梯级逻辑，需求是要实现一个这样的亮灯变化过程，如图 5-32 所示。设整个周期时间是 7s，2s 后右边的两盏灯亮，过 2s 后左边的两盏灯亮，再过 3s 后 4 盏灯全亮。

根据需求编写梯级逻辑如图 5-33 所示，这原本是早期的一道计时器的训练题，利用时基为 1s 的计时器 ACC 值的位状态来编写梯级条件的逻辑，由于 Logix 控制器没有时基为 1s 的计时器，改为计数器来做，不过，利用 ACC 值的位状态编程，在道理上是一样的。

这属于不规则的时间动作，采用 ACC 二进制的识别方式，不需要用多个计时器来完成，一个 ACC 值就可以分辨出几个时间点，并在时间点上执行相应的动作。ACC 值低三位所表达的 7s 的二进制以及在数轴上 3 个时间点的表达如图 5-34 所示。

选择任意 3 个时间点，使得它们满足 2s、2s、3s 的变化规律。完整地表达这 3 个时间点

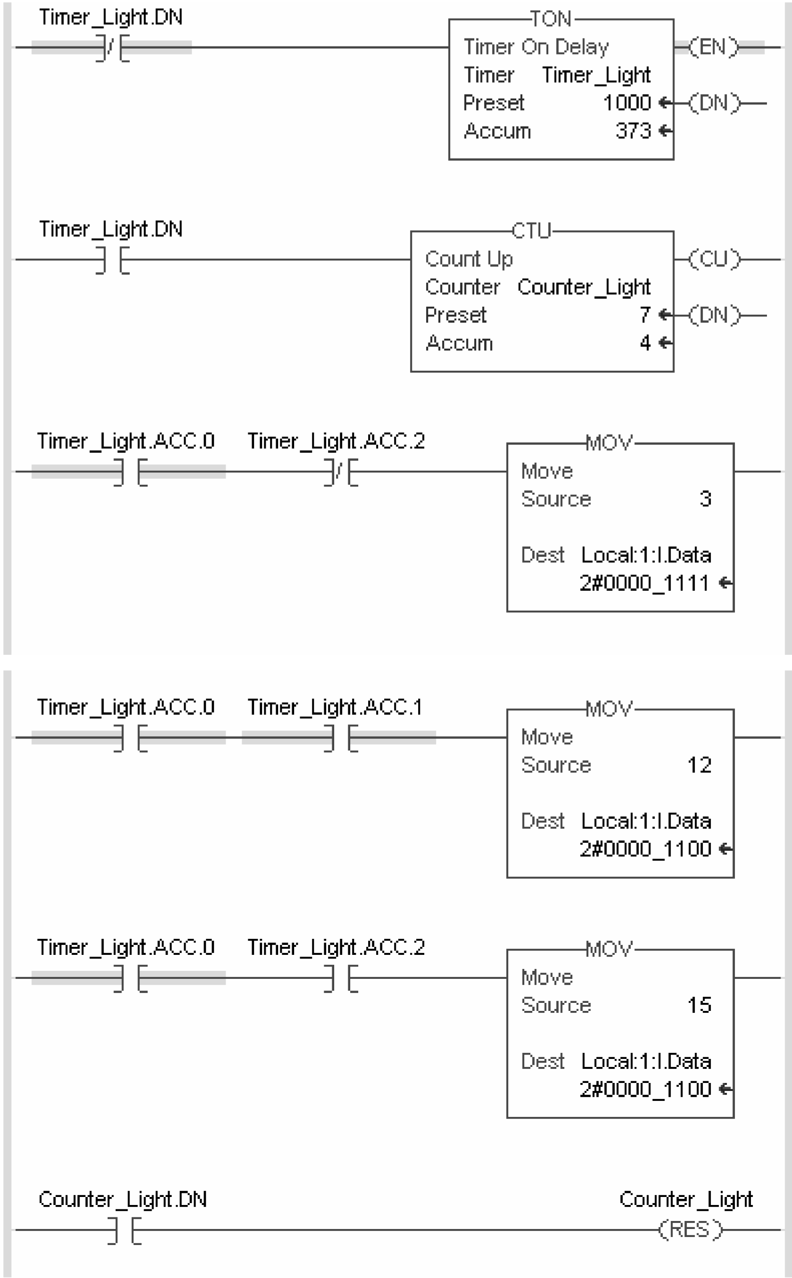


图 5-33　根据需求编写的梯级逻辑

应该是 3 位二进制位状态，考虑本例中我们只用到 3 个时间点，各用 2 位便可令它们不冲突，故使用的位状态如图 5-33 中的梯级逻辑所示。这种方式特别适合不规则的时间点操作。

点灯的动作也是可以讨论的，每次点两盏灯或者 4 盏灯，点两盏灯，用位指令设定也未尝不可，点 4 盏灯用位指令就有点累赘了，不如用 MOV 指令传送给 4 个位。

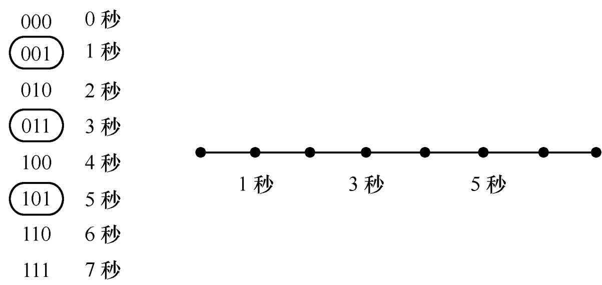


图 5-34　ACC 值低三位 7s 的二进制以及在数轴上 3 个时间点的表达

MOV 指令是直接传送立即数，可以借助于数据表的数据形式换算出来，先将一个双整字打开，改变成二进制的数据形式，将相应位置成 1 后，改为十进制的数据形式，这就是将要对 MOV 指令的目标地址传送的立即数。请注意 MOV 指令传送的立即数是十进制数。采用这样的方法，不管传送的双整字的状态位是多么的没有规律，总能换算出要传送的立即数。

顺便提及，计数器的 ACC 值的最低位，有时也用作奇偶判断，这正是二进制计算机特征的实用价值，刚才谈论的人机界面的设置实例，就是利用的奇偶关系。早期的工程师们似乎更在意我们使用的是一个二进制的计算机，对寄存器的依赖性更强，谋求了不少位状态编程的利用价值。如今的编程人员过度地依赖计算机的智能形式，反而对计算机最原始的本质视而不见，编程风格也就迥然不同了。

早期 PLC 的计数器还担任了一个重要的任务，就是用来管理文件操作指令，批量数据的处理需要设置文件的长度和记录正在执行当中的位置（称为指针），在第二代产品 PLC5/SLC500 中演变为文件操作指令专用的控制器文件，在 Logix 控制器的指令系统中，则称为控制结构数据，也就是下面章节我们将要学习的数组操作指令中用到的控制结构数据标签。控制结构数据所表现的特征跟计数器结构数据是极其相似的，乃至控制结构数据指针的增加和减少的触发也呈现了计数器的本性，充分理解计数器的工作原理，有助于我们今后学习数组操作指令时对控制结构数据的运用。

5.3　计时器和计数器联合使用的实例

计时器和计数器是经常联合使用的，自复位的计时器可以为计数器提供一个定时的计数脉冲，从而得到一个自动计数的计数器。

为配合人机界面开发软件的组态练习，用计时器和计数器编写一个模拟程序，运行在控制器中模仿现场控制状态。如图 5-35 所示，一个混合容器（Tank），上部阀门 1（Valve1）和阀门 2（Valve2）控制通过两个管道将不同液体注入混合容器，下部阀门 3（Valve3）控制通过管道流出的混合后的液体。

实例需求：

- 3 个阀门的流量可以调整；
- 既可以在本地用输入开关控制，也可以在人机界面远程控制；
- 有上溢出和下溢出限制控制，发生溢出时不能继续流入或流出液体。

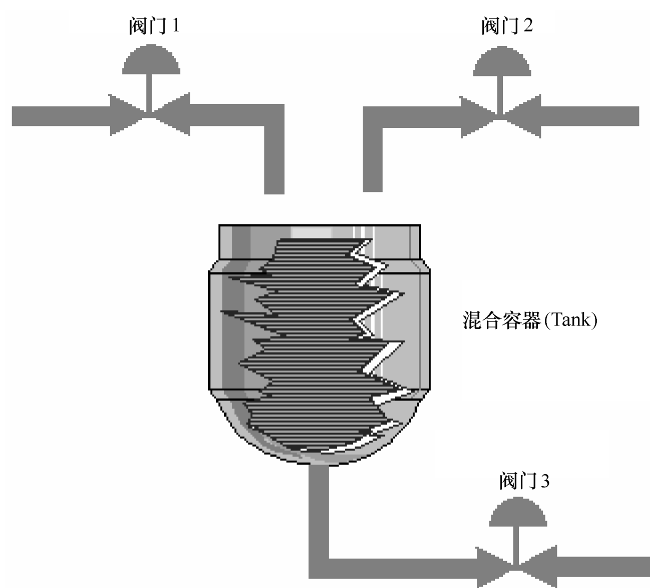


图 5-35 现场控制示意图

我们用 3 个计时器来模拟 3 个管道的液体流动，这是自复位计时器，按照设定的预置值产生计数脉冲，脉冲发送的频度，也就是模仿的液体流量。编写梯级逻辑如图 5-36 所示，3 个计时器指令的梯级来模仿液体流动。修改各个计时器的预置值，也就修改了模拟的液体流量，预置值越大，DN 置位的频度越低，流量越小；预置值越小，DN 置位的频度越高，流量越大。阀门打开时，液体开始流动；阀门关闭时，液体停止流动，阀门由本地或远程来控制。

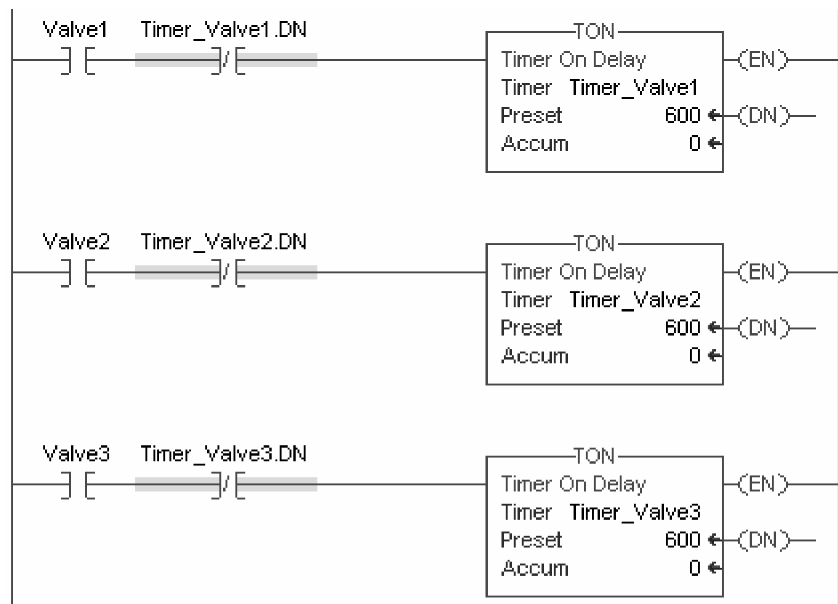


图 5-36 编写 3 个计时器来模仿液体流量

计数器的脉冲计数累积，模仿了混合容器液体平面的位置，即容量。编写梯形逻辑如图 5-37 所示，3 个计时器的 DN 作为 3 个计数器的计数脉冲，上部的两个阀门由增计数器来模拟液体流入混合容器升高液位；下部的一个阀门由减计数器来模拟液体流出混合容器降低液位。模拟上部阀门的计数器梯级被上溢出状态位限制，当达到上限位值时，液体停止流入混合容器；模拟下部阀门的计数器梯级被下溢出状态位限制，当达到下限位值时，液体停止流出混合容器。

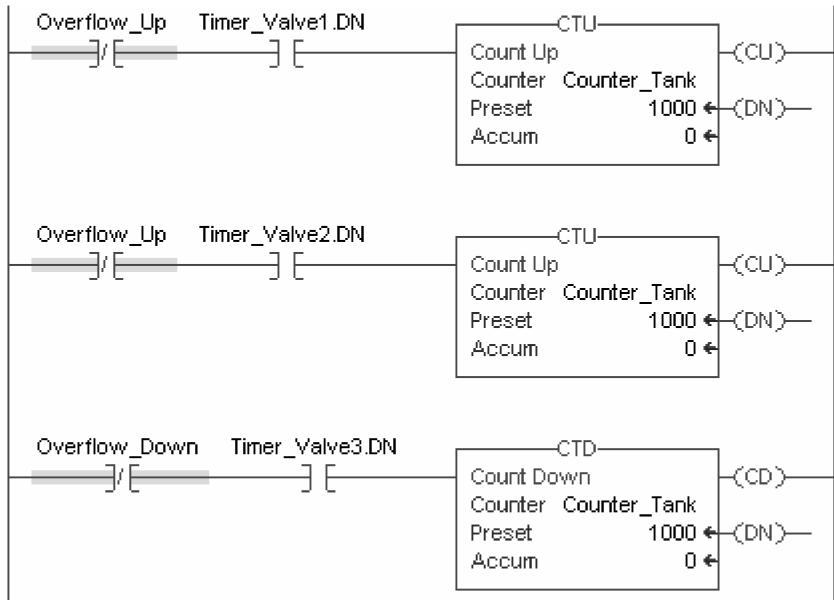


图 5-37　编写 3 个计数器来模仿容器液位变化

三条梯级看上去有点累赘，那么两个上部的计数器的梯级条件能不能并列呢？如图 5-38 所示，回答是不可以的。计数器是按梯级条件跳变来计数脉冲的，如果两个输入并列，在计数脉冲同时发生的时候，只有一个梯级条件成立，计数器只会计数一次，从而丢失了计数脉冲。特别是当计时器都是相同的预置值时，由于同步运行，丢失脉冲的几率就更大了。在此种情况下，必须分为两个梯级，即使两个计时器完成位 DN 同时置位，令前梯级 Timer_Valve1. DN 计数一次，后一个梯级 Timer_Valve2. DN 再计数一次，以保证脉冲都被记录下来，因而表达了流量的准确性。

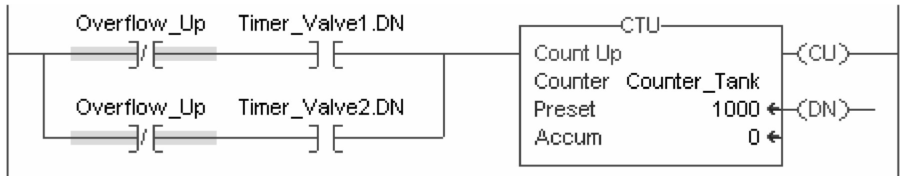


图 5-38　两个上部计数器并列的梯级条件

用两条比较指令来判断混合容器（Tank）的上溢出和下溢出，从而限制上部的增加或下部的减少。对于模拟程序来说是必不可少的，因为计数器可以出现负数或高于混合容器（Tank）液面的最大值，模拟的混合容器（Tank）却不可以，实际操作也是如此，容器都是

有上限位和下限位的检测和管理的。梯级逻辑编写如图 5-39 所示，比较指令产生的上限和下限输出报警状态位 Overflow UP 和 Overflow Down，一方面用来限制计数器计数的梯级条件，令它不能继续增加或继续减少，相当于关闭了阀门；另一方面为人机界面（HMI）提供所需要的报警状态显示，告知阀门的控制已经失去作用。

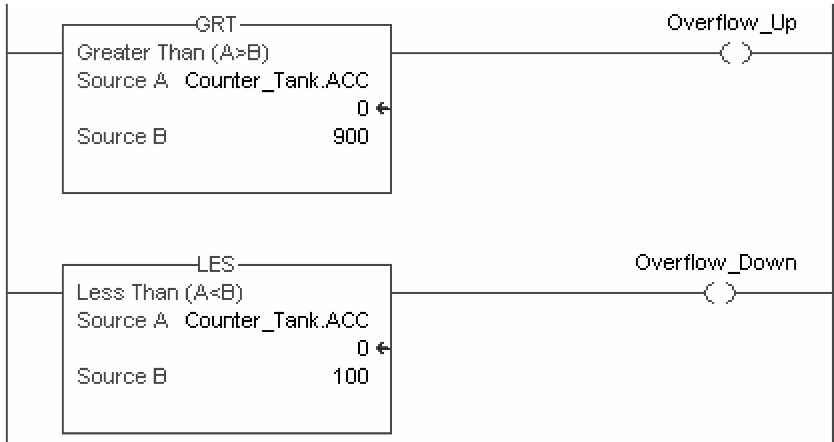


图 5-39　上限位和下限位的检测

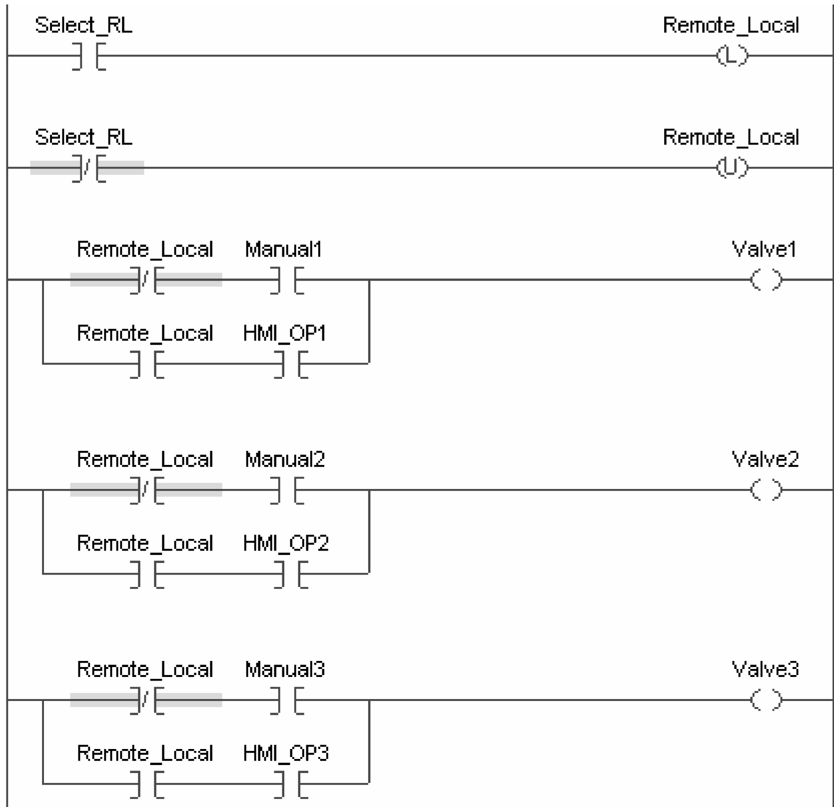


图 5-40　本地或远程控制阀门

现在来看阀门是如何被控制的。编写梯级逻辑如图 5-40 所示，每个阀门都有两路控制，一路是本地的手动控制，另一路是远程的人机界面（HMI）上的操作来控制。这两路选择由选择开关控制，选择开关打向远程，远程一路接通，远程控制有效；选择开关打向本地，本地一路接通，本地控制有效。本例中规定，Remote_Local 为 1 远程有效，为 0 本地有效。这是远程本地必居其一的做法，并对远程或本地做出明显的选择。本地或远程的选择 Select_RL，可以由一个位置开关来担任，也可以在人机界面上来操作。

也有例外，如果对远程或本地没有特别的限制，任何一个操作都会引起回归，可以按如图 5-41 所示编写的梯级逻辑，这样如果原先的状态在本地操作，只要一个远程操作，就把远程操作抢了过来，如果再来一个本地操作，又把本地操作抢了过来。这种做法使得远程本地操作切换便利，几乎无须额外的介入，适合远程本地操作是随意的情况，但是很多控制系统是本地优先的，一旦处在本地操作，远程是不可以介入的，否则会有一定的风险。

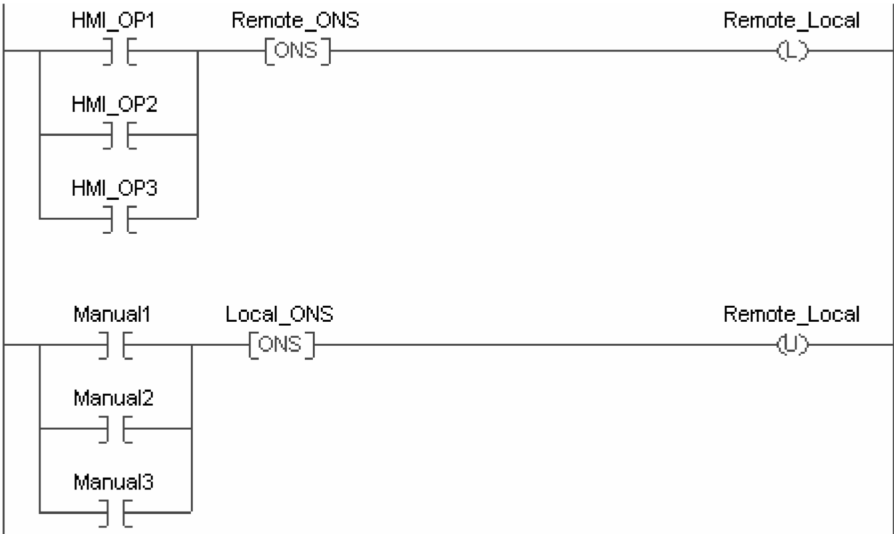


图 5-41　本地或远程选择

以上梯级逻辑的编写是按照编程过程的思路来讨论的，实际上，我们在编写梯级逻辑时，应该按照执行过程的先后插入或加入梯级，所以刚才的编写顺序并不是梯级逻辑执行的顺序。如图 5-42 所示才是执行例程中完整的梯级逻辑的执行顺序。

下载到控制器运行测试，一切正常。

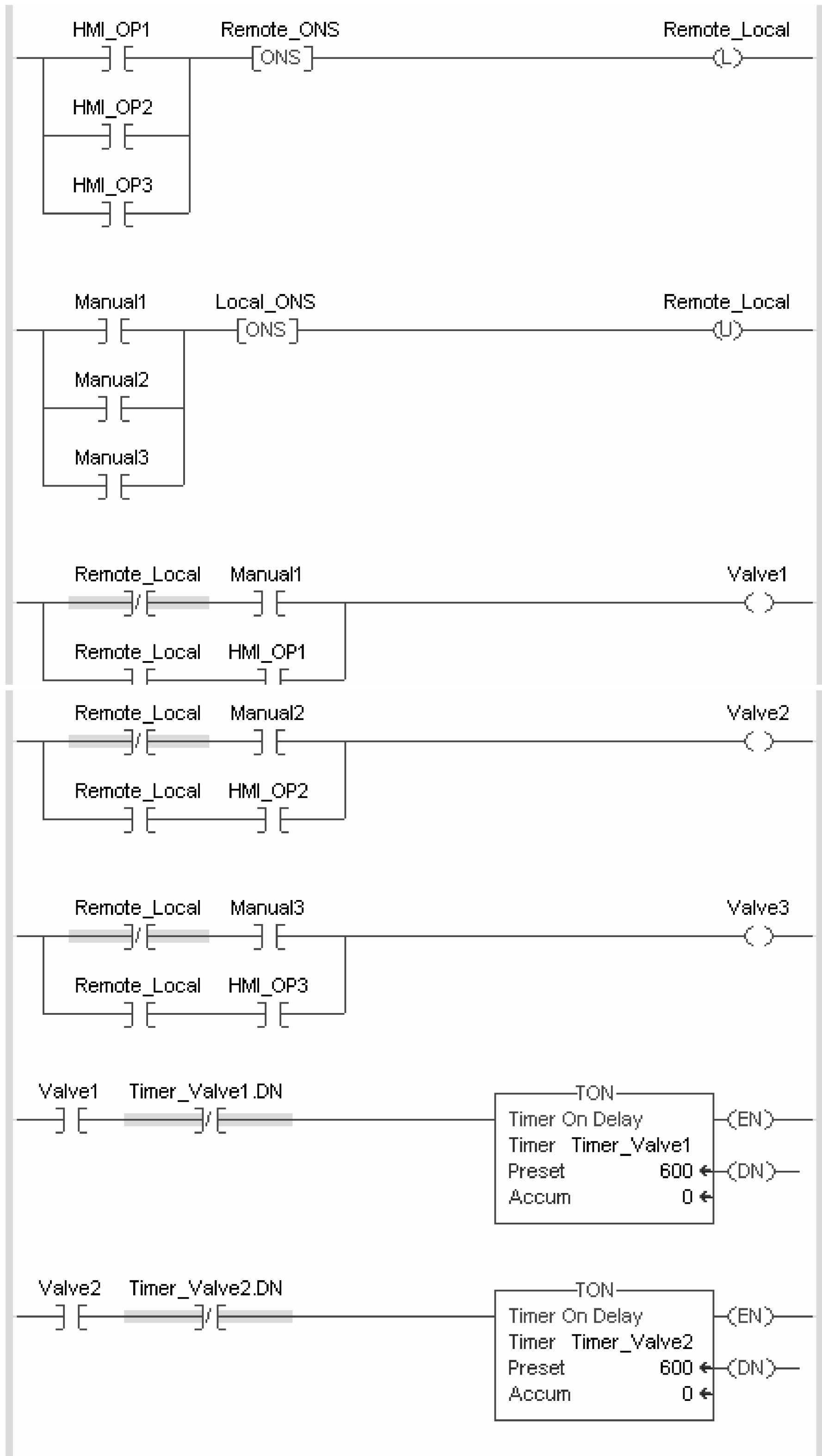


图 5-42 完整的梯级逻辑执行顺序

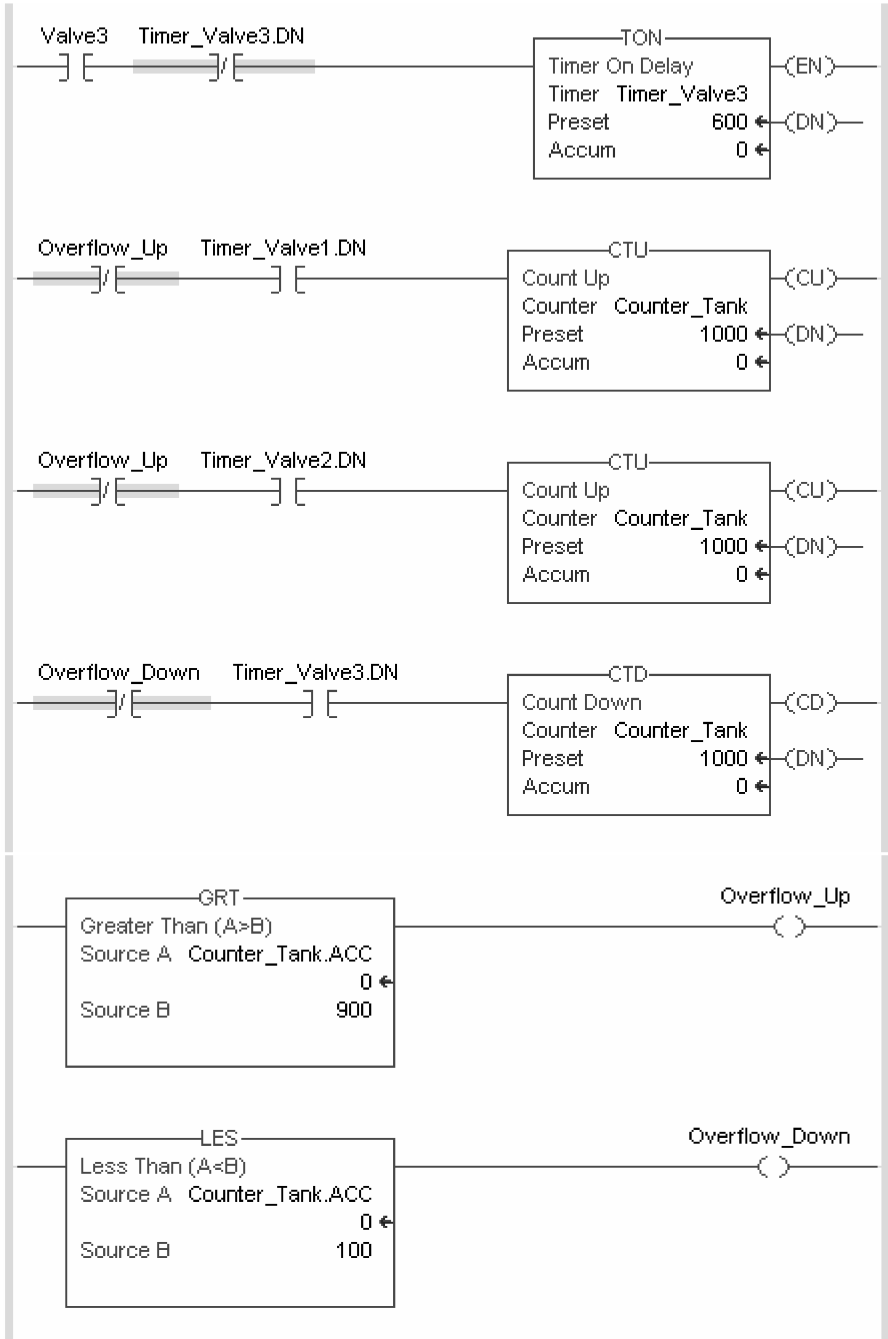


图 5-42　完整的梯级逻辑执行顺序（续）

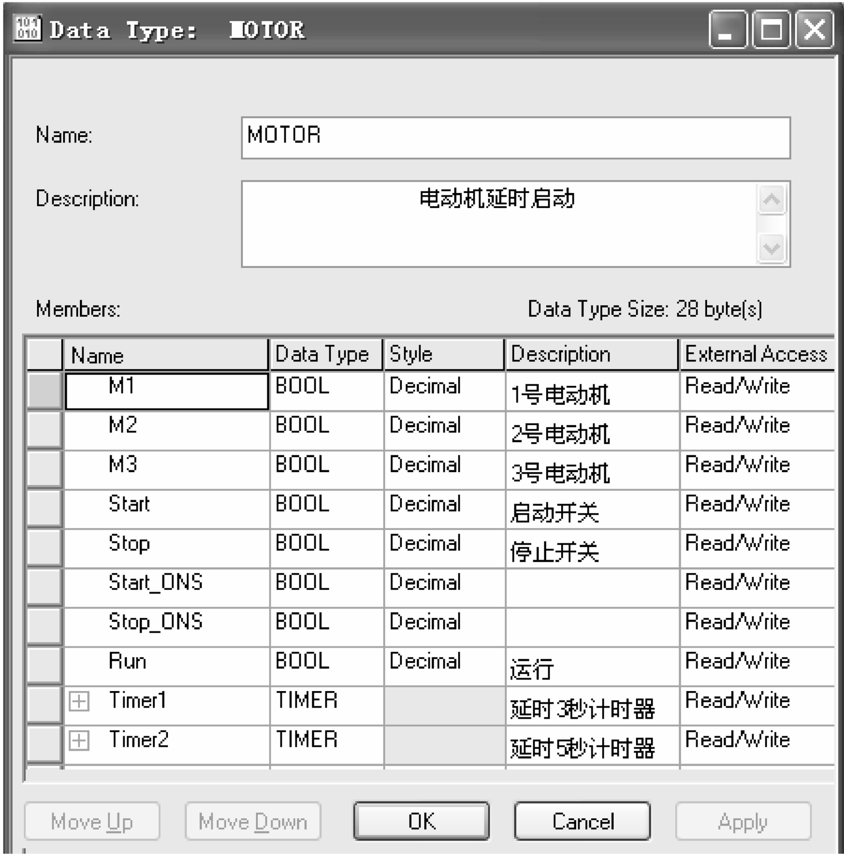
5.4　采用用户自定义结构数据编程

上面我们学习编写了一些梯级逻辑，都是直接引用数据标签。在 Logix 控制器中一个最大的优势就是用户自定义数据结构标签可以围绕一个控制对象，将不同类型的数据集合在一起，建立相应的结构数据标签。特定的结构数据标签，不管是编程还是监视数据，都非常的

方便，这正是 Logix 控制器的一大特色。我们来尝试一下，仍然是有关计时器的指令编写梯级逻辑，但使用用户自定义数据结构标签来编写。

实例：有一个延时启动控制的需求，当开关启动时，第一个电动机起动，延时 3s 后，第二个电动机起动，延时 5s 后，第三个电动机起动。开关关闭时，所有电动机复位，起动尚未完毕，开关关闭，所有电动机亦复位。编写梯级逻辑来完成这个过程，并使用用户自定义结构数据标签。

创建名为 MOTOR 的用户自定义结构数据，如图 5-43 所示，围绕电动机这个控制对象，创建相关的子元素，并给予说明。我们分别为 3 个电动机创建了 M1、M2 和 M3 的 3 个布尔量标签；为启动和停止创建了 Start 和 Stop 的两个布尔量标签；创建 Timer1 和 Timer2 的两个计时器结构数据标签，以及用于 ONS 指令存储位的两个布尔量标签，内嵌在用户自定义结构数据中的 ONS 存储位，紧凑地利用了现有的空间。在编程过程中，如需要添加新的子元素，可以为结构数据添加新的元素定义，创建的标签也会随之跟着添加。



Name	Data Type	Style	Description	External Access
M1	BOOL	Decimal	1号电动机	Read/Write
M2	BOOL	Decimal	2号电动机	Read/Write
M3	BOOL	Decimal	3号电动机	Read/Write
Start	BOOL	Decimal	启动开关	Read/Write
Stop	BOOL	Decimal	停止开关	Read/Write
Start_ONS	BOOL	Decimal		Read/Write
Stop_ONS	BOOL	Decimal		Read/Write
Run	BOOL	Decimal	运行	Read/Write
Timer1	TIMER		延时3秒计时器	Read/Write
Timer2	TIMER		延时5秒计时器	Read/Write

图 5-43　创建用户自定义结构数据 MOTOR

在标签数据库创建结构数据标签 MotorStart，如图 5-44 所示，其数据类型选择 MOTOR。可以看到，结构数据子元素的说明关联到了新建的数据标签中，以浅色显示。用户自定义结构数据定义中的说明是可以延伸到每一个标签的，这无疑节省了内存，如果标签中对个别子元素要额外地予以说明，这个说明是可以覆盖原结构数据子元素说明的，不过是另外消耗内

⊟ MotorStart	MOTOR	电动机延时启动
MotorStart.M1	BOOL	电动机延时启动 1号电动机
MotorStart.M2	BOOL	电动机延时启动 2号电动机
MotorStart.M3	BOOL	电动机延时启动 3号电动机
MotorStart.Start	BOOL	电动机延时启动 启动开关
MotorStart.Stop	BOOL	电动机延时启动 停止开关
MotorStart.Start_ONS	BOOL	电动机延时启动
MotorStart.Stop_ONS	BOOL	电动机延时启动
MotorStart.Run	BOOL	电动机延时启动 运行
⊞ MotorStart.Timer1	TIMER	电动机延时启动 延时3秒计时器
⊞ MotorStart.Timer2	TIMER	电动机延时启动 延时5秒计时器

图 5-44　在数据库创建标签 MotorStart

存而已，其显现的是重色而不是浅色。

根据电动机启动的需求，编写梯级逻辑如图 5-45 所示，并运行测试。

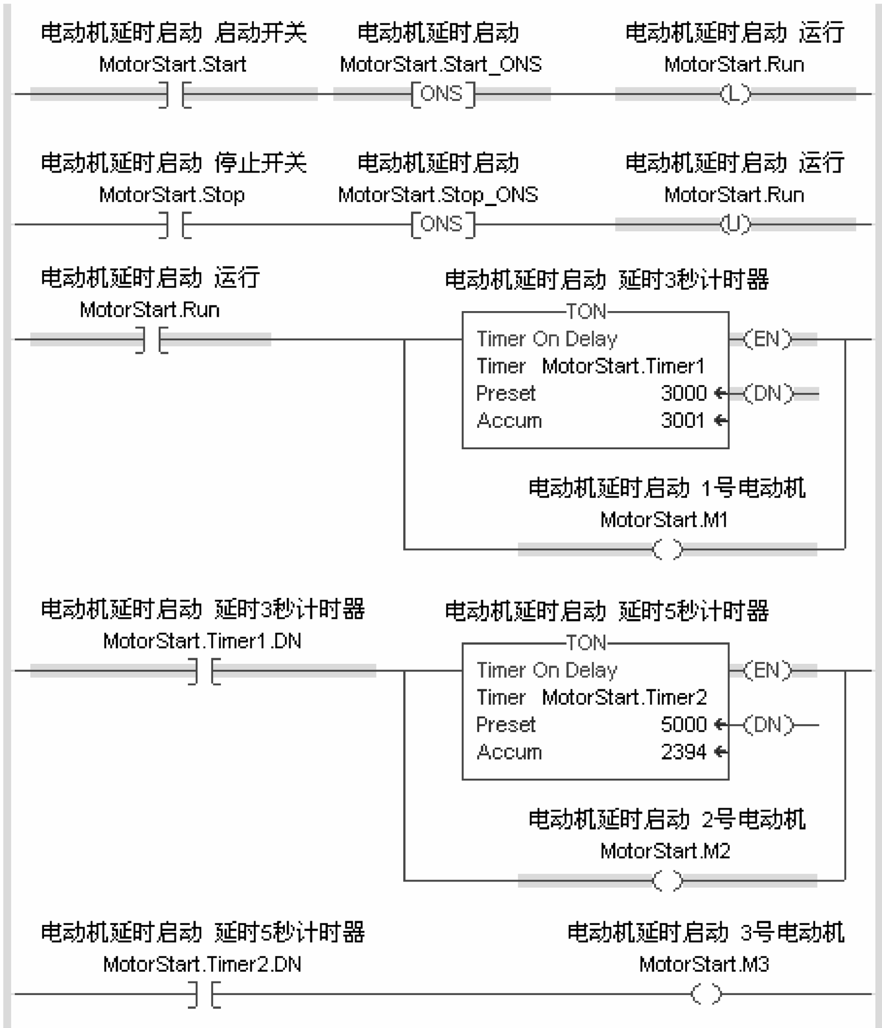


图 5-45　电动机启动过程的梯级逻辑

电动机启动开关 MotorStart. Start 和停止开关 MotorStart. Stop 是分别独立工作的，ONS 指令限制了梯级条件一次操作有效，运行位 MotorStart_Run 用来控制整个延时启动过程，当启动开始，运行位提供了计时器 MotorStart. Timer1 的梯级条件并启动 1 号电动机 MotorStart. M1，3s 之后计时器 MotorStart. Timer1 完成计时，MotorStart. Timer1. DN 位为计时器 MotorStart. Timer2 提供了梯级条件，同时启动 2 号电动机 MotorStart. M2，5s 后，3 号电动机 MotorStart. M3 启动。

在启动的过程中，假如尚未完成便停止启动，可以看到所有的梯级条件随之消失，消失的梯级条件帮助计时器复位，启动不会终止在起动中途的状态，而是处于复位状态，为下次启动作准备，从而回到了原点。

用户自定义结构数据给我们带来的好处之一是可以在一个结构数据标签中，看到一个控制对象所有的相关数据，一目了然。用户自定义结构数据，往往是出于不同的目的创建，有时为了同一控制对象，有时为了通信数据传送，有时为了组态数据模块，总是将有关联的数据集合在一起，使得应用更为方便或获得更好的系统性能。如图 5-46 所示的数据，就是我们刚才的梯形图例程中使用数据的监视页面，是针对一个同一控制对象而创建的，相关的计时器和输出都集合在一起。

⊟ MotorStart	{...}
MotorStart.M1	1
MotorStart.M2	1
MotorStart.M3	1
MotorStart.Start	1
MotorStart.Stop	0
MotorStart.Start_ONS	1
MotorStart.Stop_ONS	0
MotorStart.Run	1
⊟ MotorStart.Timer1	{...}
⊞ MotorStart.Timer1.P...	3000
⊞ MotorStart.Timer1.A...	3001
MotorStart.Timer1.EN	1
MotorStart.Timer1.TT	0
MotorStart.Timer1.DN	1
⊟ MotorStart.Timer2	{...}
⊞ MotorStart.Timer2.P...	5000
⊞ MotorStart.Timer2.A...	2394
MotorStart.Timer2.EN	1
MotorStart.Timer2.TT	1
MotorStart.Timer2.DN	0

图 5-46　结构数据标签 Motor Start 的数据监视

第 6 章

算术逻辑运算指令的编程

控制器的指令系统中，算术逻辑运算指令是最接近 CPU 汇编语言的，它让人一看便知，根本不需要对指令进行任何解释。早期那些让人肃然起敬的工程师们，为了节约内存，或针对控制过程的不同数据模式，不惜绞尽脑汁，编出许多匪夷所思的梯级逻辑，这都是彻头彻尾的技巧性编程。希望功能强大的 Logix 控制器的充裕资源和数据容量能让我们化繁为简。

算术运算指令又与生产过程紧密相关，尤其是有关的数学模型不是一个初来乍到的人一口气就能读懂的，常常对着一大段运算梯级逻辑一筹莫展，所幸这样的尴尬逐步被黑匣子式的 Add_On 指令所囊括，演变为模块化的功能。一旦在解读例程时遇到这样的障碍，我的意见是暂且绕过，而去追踪那些执行进程。

6.1　一般算术运算指令编程

和通常的算术运算完全一样的运算过程，并遵循所有的算术运算规则，控制器系统完成一般算术运算的指令如下所列：

- ADD　加法指令
- SUB　减法指令
- MUL　乘法指令
- DIV　除法指令
- CPT　表达式运算指令
- MOD　求模运算指令
- SQR　求平方根指令
- NEG　求反指令
- ABS　求绝对值指令

为了了解算术指令以及配合相应的操作，下面我们编写一段梯级逻辑来计算例程运行的平均时间，尽管我们可以在任务的监视页面看到任务扫描所有例程的当前瞬间时间和历史最大时间，如果我们想综合地评价控制系统程序响应的性能，对例程扫描的平均耗用时间，不妨精确地计算一下，编写梯形逻辑如图 6-1 所示。

启动按钮开始整个累加过程，按钮操作令位标签 Start_Cyc 置位，锁定运行位 Run_Cyc，累加扫描次数工作开始。运行位 Run_Cyc 作为计时器的梯级条件，计时器 Timer_Cyc 开始启动，计时器设定的预置值就是累加运算的时间段，此处设置的时间是 1min。

但凡用到加法指令 ADD 作为累加器，即加 1 送回原数据标签，一定要留意它的梯级条

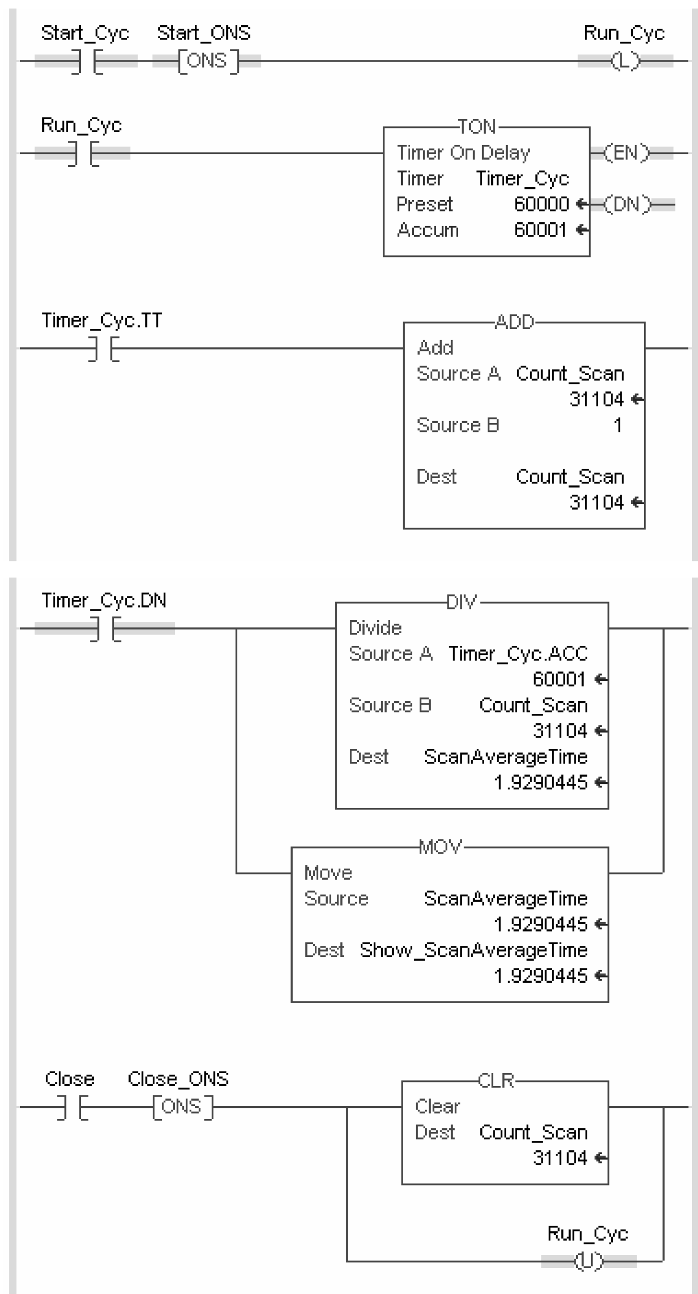


图 6-1　编写梯形逻辑

件，每当扫描时梯级条件成立，指令将完成相加的运算，即累加操作。本例中是计算当前例程扫描次数，梯级条件是计时器的计时位 TT，在计时器正在计时的时间段，每次例程扫描都会加 1，直到计时器完成位置位。

计时器的完成位 DN 置位时计时位 TT 复位，ADD 指令不再累加扫描次数，用计时器的完成位 DN 来作为计算例程扫描的平均时间的梯级条件，除法指令 DIV 计算出平均扫描时间，注意这个时间单位应该是毫秒。显然，决定累加时间段的计时器预置值影响了计算平均

值的精度，这个时间段越大，就越接近准确的平均值，这里选择的 1min 就足够了。完成后计算出结果，紧接着除法指令之后，用 MOV 指令将计算结果送到用于显示的通信结构数据标签中，这几乎是一种习惯，一般不会直接将计算结果用来显示，而是转至缓冲区，这样可保持数据的完整和稳定。

最后，关闭动作实施善后工作，清除累加器 Count-Scan，复位运行位 Run_Cyc，以准备下一次的累加计算工作。

这里我们用到两条算术运算指令，这样的计算基本上是能读懂的，用的都是指令的基本功能，我们只要留意它们给予的梯级条件就可以了。

除了加、减、乘、除等完成单一运算的指令，还有一条综合运算指令 CPT，可以通过运算表达式完成复杂的运算，是应用最为灵活的运算指令。此处的运算表达式以及控制器在任何地方的运算表达式的书写，都遵循加减乘除的运算规则。如图 6-2 所示的梯级逻辑，CPT 指令的表达式用来计算比例系数，如果用单一的运算指令，则需要多个分散的梯级来完成，CPT 指令可以集中地体现某种运算关系，令人一目了然。

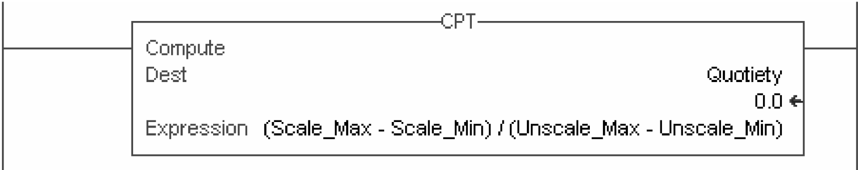
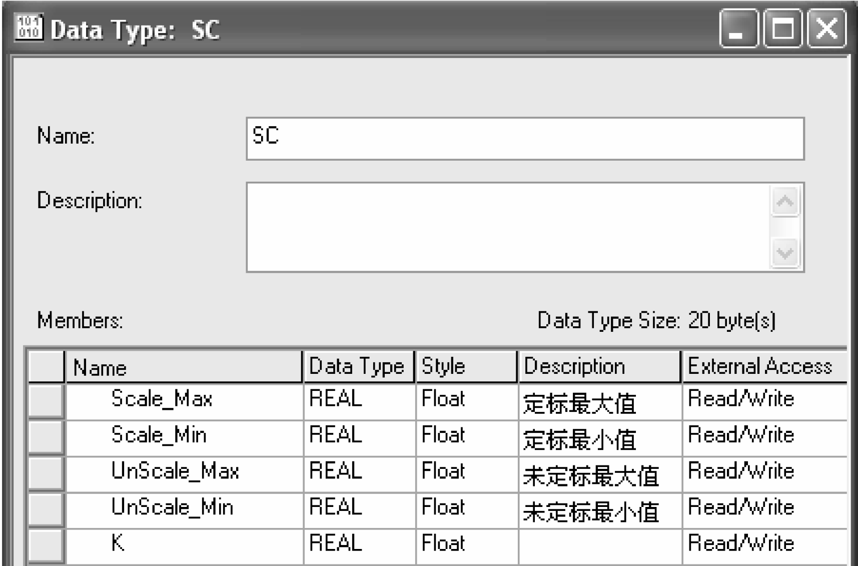


图 6-2　综合运算的梯级逻辑

下面我们运用 CPT 指令来换算模拟量输入通道的定标值。我们知道，1769-IF4XOF2 模块自身是能不定标的，需要编程来定标处理，假定我们要将模块的 0～31140 的数据范围。换算为我们后面多次要运用的定标范围 0～10000。我们先创建一个用户自定义结构数据，以便编写的梯级逻辑数据标签更为紧凑和集中，并能直接看出几个梯级运算之间的关系，如图 6-3 所示。



Name	Data Type	Style	Description	External Access
Scale_Max	REAL	Float	定标最大值	Read/Write
Scale_Min	REAL	Float	定标最小值	Read/Write
UnScale_Max	REAL	Float	未定标最大值	Read/Write
UnScale_Min	REAL	Float	未定标最小值	Read/Write
K	REAL	Float		Read/Write

图 6-3　创建一个用户自定义结构数据

再在数据库中建立相应的标签，如图 6-4 所示。

⊟SC	{...}		SC	
SC.Scale_Max	0.0	Float	REAL	定标最大值
SC.Scale_Min	0.0	Float	REAL	定标最小值
SC.UnScale_Max	0.0	Float	REAL	未定标最大值
SC.UnScale_Min	0.0	Float	REAL	未定标最小值
SC.K	0.0	Float	REAL	

图 6-4　在数据库中建立相应的标签

最后，编写梯级逻辑如图 6-5 所示。用传送立即数来设定未定标范围和定标范围，然后用 CPT 指令计算换算的比例系数，最后通过乘法指令得到定标后的模拟量输入。用户自定义结构数据让人看到几个梯级逻辑中相关的运算关系，未定标范围和定标范围的数据可以在数据表中直接设置，但是如果用程序设置，就非常可靠了。基于系统优化的考虑，减少梯级逻辑扫描，只作初始化设置也是常见的，即使数据意外被改动，只要重新运行程序，正确的数据就会被设置。

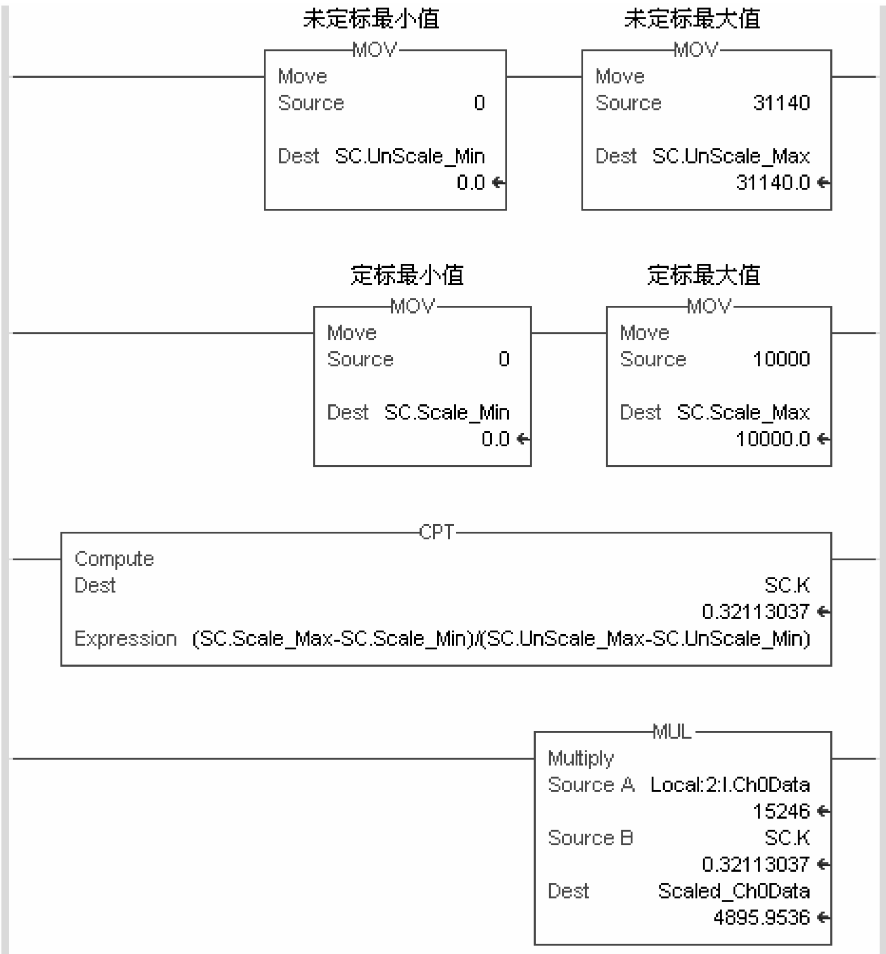


图 6-5　定标运算的梯级逻辑

CPT 指令运行较耗费时间，同样完成一个单一的运算，CPT 指令的执行要消耗更多的时间，所以完成单一运算，建议用单一运算指令，比如单一相加的操作就直接用 ADD 指令而不要用 CPT 指令的表达式来做，如图 6-6 所示的梯级逻辑就不建议采用。

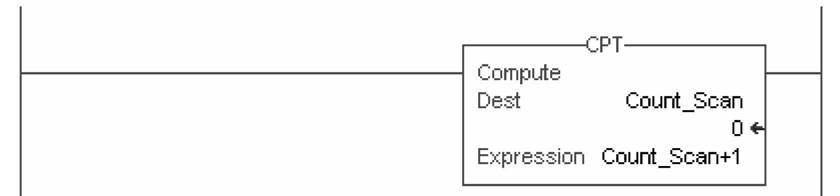


图 6-6　不建议采用的梯级逻辑

6.2　算术指令编程实例

我们介绍两个现场实例，来使用算术指令编写梯级逻辑。

我曾经到过一个自来水厂，那里的操作工抱怨他们的每日产量记录不准确。我调看了一段流量积累的例程，梯级逻辑自然是编得没有道理。我借用这个实例，编写了一个用算术指令处理的例程，作为教学使用，在这里跟大家分享一下。

实例的需求是：

一个采集流量的传感器，按吨/小时采集，传感器提供的是模拟量信号，且不管它是电流信号还是电压信号，作为精度足够高的模/数转换，我们得到的模拟量输入数据至少是千位数的，如果定标太低，精度就牺牲太多了。现在，我们要将这个采集到的数据进行累积，以得到每天流出自来水厂的总量，也即自来水的产量。

首先要分析的是每隔多长时间采集一次数据可以保证积累正确，有关人员提供了可靠的依据，对水量来说，1s 采集 1 次数据足以保证精确。现在我们用一个计时器就可以解决问题了，每秒钟产生一个动作脉冲，如图 6-7 所示。Logix 控制器的计时精度足够满足这个需求，如果要求更紧密、更精确的时间采集，我们只好求助于定时中断任务了。



图 6-7　计时器每秒钟产生一个动作脉冲

下面要解决的是如何积累数据，用一条 ADD 指令，每秒钟对采集的数据相加一次，然后送回作为累加器的存储单元，因为我们明了这个自复位计时器的 DN 位置位能够且只能够存在一个扫描周期，所以很放心 ADD 指令 1s 只能相加一次，不会更多，如图 6-8 所示。

现在比较纠结的问题出现了，如何将流量转为容量呢？大家应该没有忘记，这个数据是一个吨/小时的信号采集来的，而数据又在按每秒积累。这时大家不约而同地想到了那个数字 3600，这是小时和秒的换算。你可能会说，这很简单，把采集积累的数据除以 3600，得到的容量单位就是吨，开始我也这么想来着。

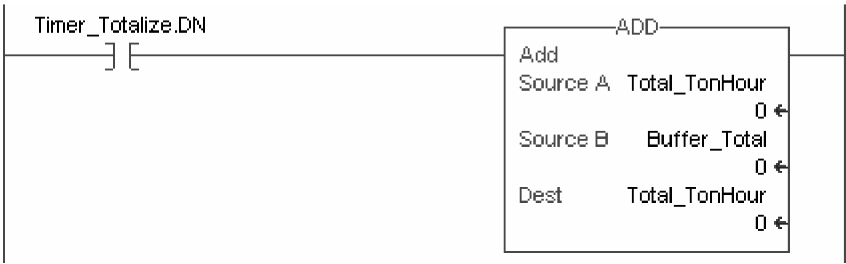


图 6-8 每秒一次的流量累加

下载到控制器的程序一运行就报错溢出了！解释一下，当时我用的是 PLC5，那是 16 位寄存器的内存单元，可以想想，16 位的最大数据存储是 32767，千位数的采集积累十几秒后就溢出了。当然 Logix 控制器提升到了 32 位的存储容量，可容纳的数据范围达到了 2147483647，存放一天的产量累积绰绰有余，不会发生溢出。如果我想要更高的精度，Logix 控制器的模拟量模块早就不是千位数的精度了，如果定标在万位数，这个容量就可能不够存放一天了，更何况存储的未必是一天的累积量，即使是 Logix 控制器 32 位的数据容量，依然面临不够存放的困境。其实我们一直在纠结数据单元的容量够不够，就是因为我们一直想用除法解决这个换算的问题。不妨换个思路试一下。

于是我编写了这样的梯级逻辑来解决换算的问题，如图 6-9 所示。其实除法的本质就是累减法，完全可以用减法来代替除法。

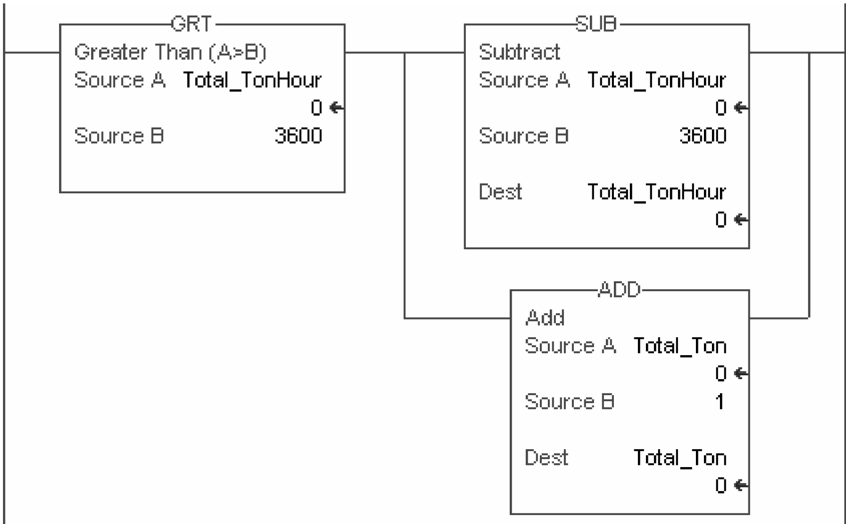


图 6-9 累减法代替除法

用一条比较指令来决定这个操作，当累积量大于 3600 时，就从累积量中减去 3600，同时在设定的以吨为单位的累积器中加 1，这样我们永远也不要担心累积量是否会溢出。我们得到结果是相当准确的，除了有不到一吨的延时残存，那是累积不到 3600 的那一部分余值。

在一个高速运算的系统中，乘除法未必优于加减法！何况计算机中的乘除法就是加减法的累积，即使我们使用了除法指令，计算机也会换成减法来运算，我们不过是回到最基础的处理模式罢了。

现在我们来讨论另外一个问题，究竟是先把流量换成吨再累积，还是先累积再换成吨

呢？如果我们采用了除法现将流量换算成吨的容量，然后再进行累积，就不可避免的会有累积误差；如果先累积，再换算成我们想得到的容量，我们又会担心累积容量溢出的问题。如采用累减法解决问题，两种忧虑都没有了。

我们将编程设备的模拟量通道 1 的模拟数据当做流量采集，编写如图 6-10 所示的梯级逻辑，乘法指令直接使用了定标比例系数 SC. K，请注意梯级的顺序，并运行测试通过。

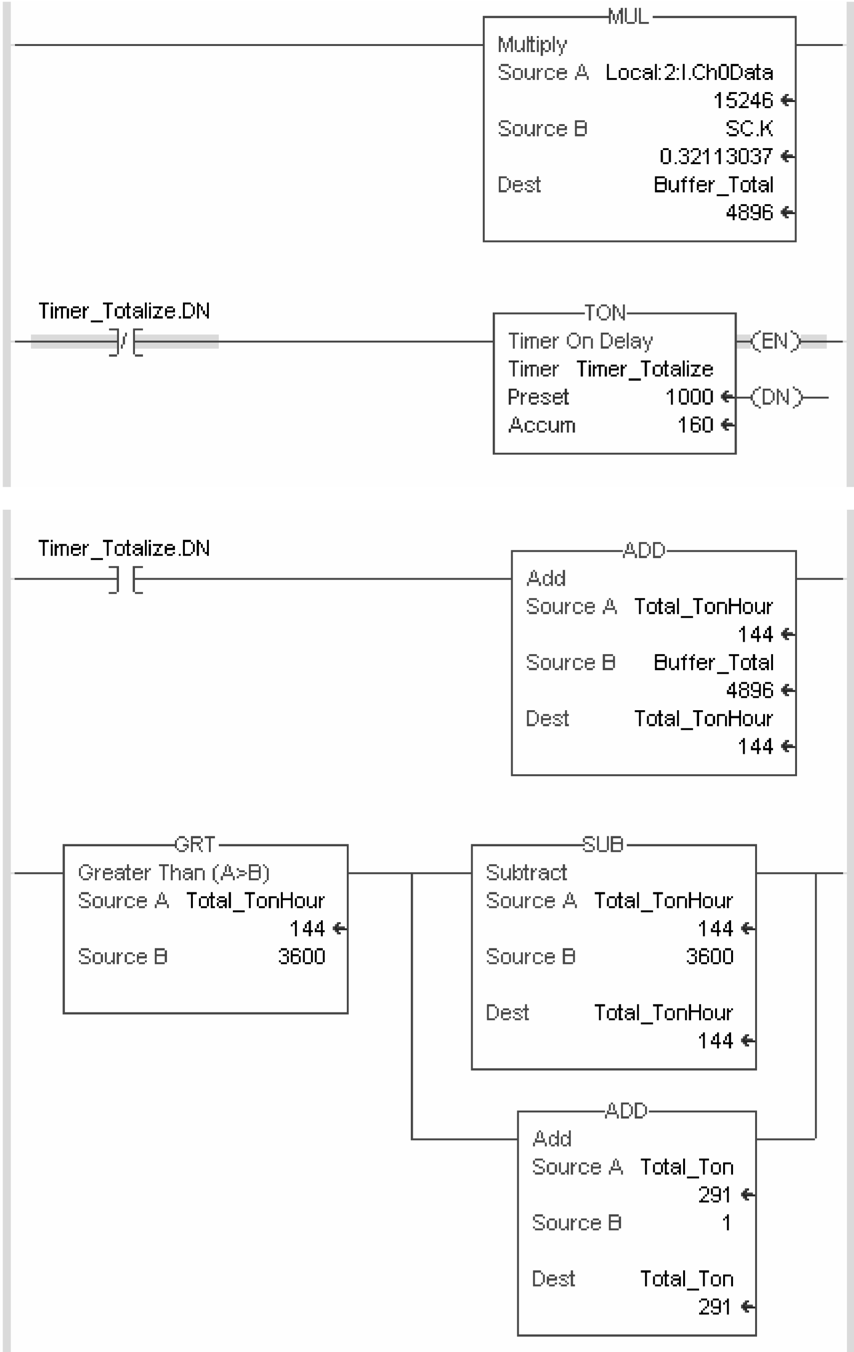


图 6-10　处理流量累积的梯级逻辑

这里我们延伸讨论一下关于数据处理环节的问题。我们知道，整个系统的信息流动，在不同的处理环节，是不同的载体、不同的表达形式和不同的数据范围。例如一个温度被采集到人机界面显示，所经历的信息环节变换过程是：

物理量 温度（范围 -40 ~ +60 ）→ 物理电信号 模拟量 4 ~20 mA → 数据 0 ~31140 → 定标数据 0 ~10000 → 换算 -40 ~ +60 →人机界面显示。

为什么在模拟量定标时不直接定为 -40 ~ +60 呢？这样做无疑影响了 A/D 转换的精度。所以我们宁愿对模拟量模块定标较大的数据范围，以获得更好的 A/D 换算精度，然后再换算到温度范围。在控制器中的每一个数据处理环节，我们考虑的是怎样获得更高的精度和最少的积累误差，而不是急于还原数据的原来面目。在使用算术指令时最需要顾虑的，不仅仅是如何获得运算的正确结果，还需考虑如何避免精度的损失。

上面所讨论的问题，还有一个地方有遗漏，实际上我们得到的数据单位并不是吨，因为从吨/小时传感器的电压或电流信号到模/数转换后的数据，还有一个换算关系需要核准，我们将最后得到的数据乘上这个核准的系数，才是真正的单位吨。我们先考虑数据处理中确保精度的最佳方案，最后再考虑换算关系。在数据信息的传递过程中，未必每个环节都是数据的真实面目。所以，不要急于将数据换算成真实的面目。

我们再来看一个例子，前面第 5 章中，有一个关于发动机运行时间累计的例子，其累计值将为机组保养提供依据，这个时间的累计值是需要作后续的数据处理的，因为需求是当发动机运行若干时间后，要给出保养的提示信息，这个时间要求以小时为单位，并以此判定。我们先假定发动机运行 3000h 后，要求触发提示信息，梯级逻辑编写如图 6-11 所示。

比较指令对发动机运转速度值进行判断并决定梯级条件，当速度达到并超过额定值时，认定发动机处在正常运转状态，比较结果令梯级条件成立，保持型计时器开始计时；当发动机运转速度低于比较值时，梯级条件不成立，保持型计时器停止计时累积，并且累加值保持不变。如此重复，计时器的累加值可能是断断续续记录了时间的积累，由于计时基值是毫秒，所以这是毫秒为单位的时间积累。

类似前面自来水流量的累积做法，为了避免累积误差，宜采用累减法来解决，每当计时器的累加值大于 3600000ms，也即 1h 的时间，便从计时器累加值中减去 3600000，同时给运行时间累积值 RunTimeTotalize 增加 1，以累积小时单位的累积器随时提供了发动机运转的累积时间以供参考。可以看出来，累积器的时间单位是可以调整的，取决于你希望用怎样的单位去积累时间，改变比较值和相减的量便改变了时间的单位。此处比较值和减去的数值都是 3600000，所以运行时间累加器所表达的数值是小时，即积累时间标签 RunTimeTotalize 的单位是小时。

当发动机保养完毕，即将开始新一轮的时间累积，可以将计时器累加值和小时累积值清零，并将提示信息复位，以便下次重新开始，梯级逻辑编写如图 6-12 所示。正如所有实际例程所做的那样，附加于算术运算指令信息处理的梯级逻辑也是必不可少的。

一段完整的数据采集、数据处理和复位操作的梯级逻辑，如图 6-13 所示，并运行测试通过。

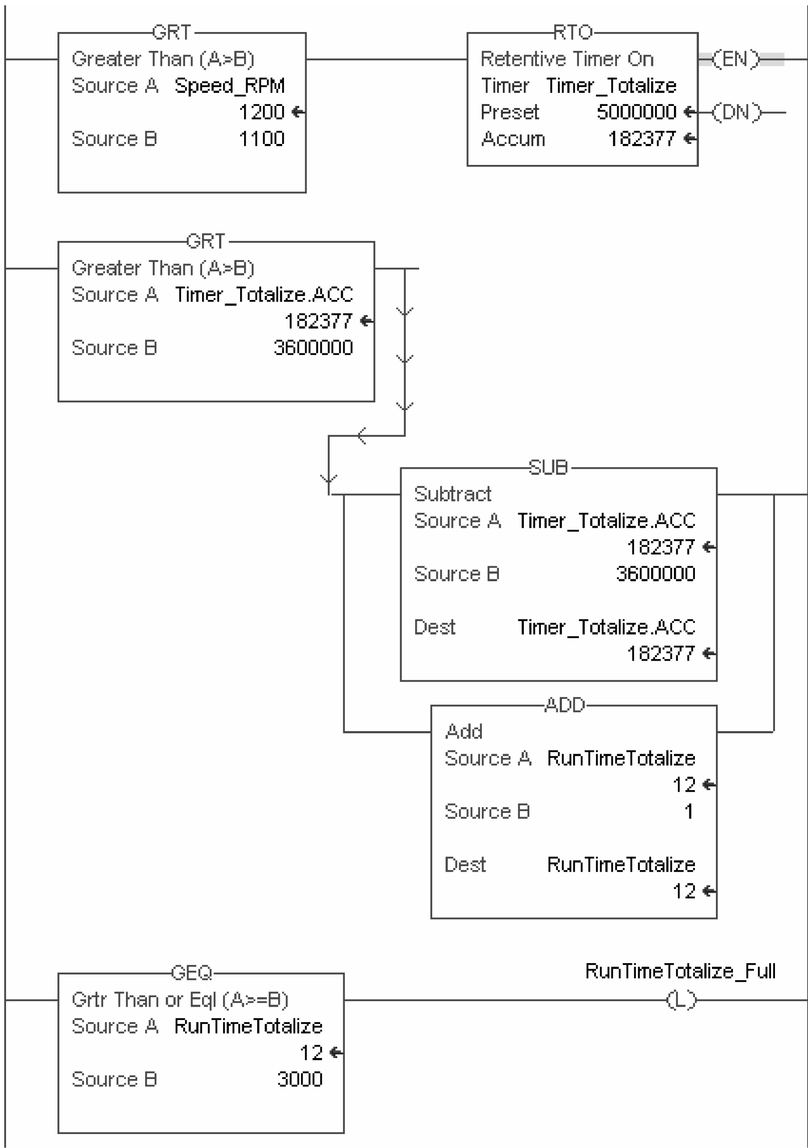


图 6-11　发动机运行时间累积

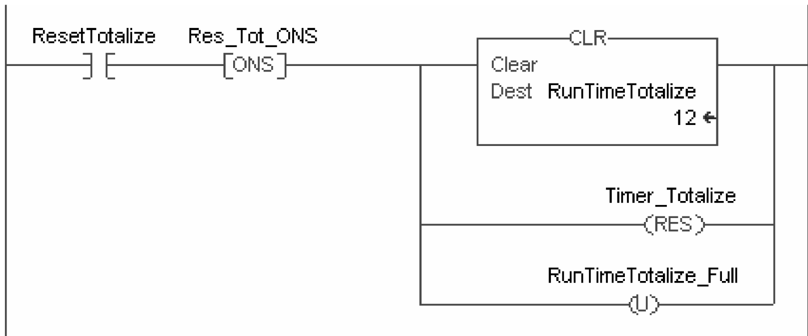


图 6-12　复位所有数据和状态

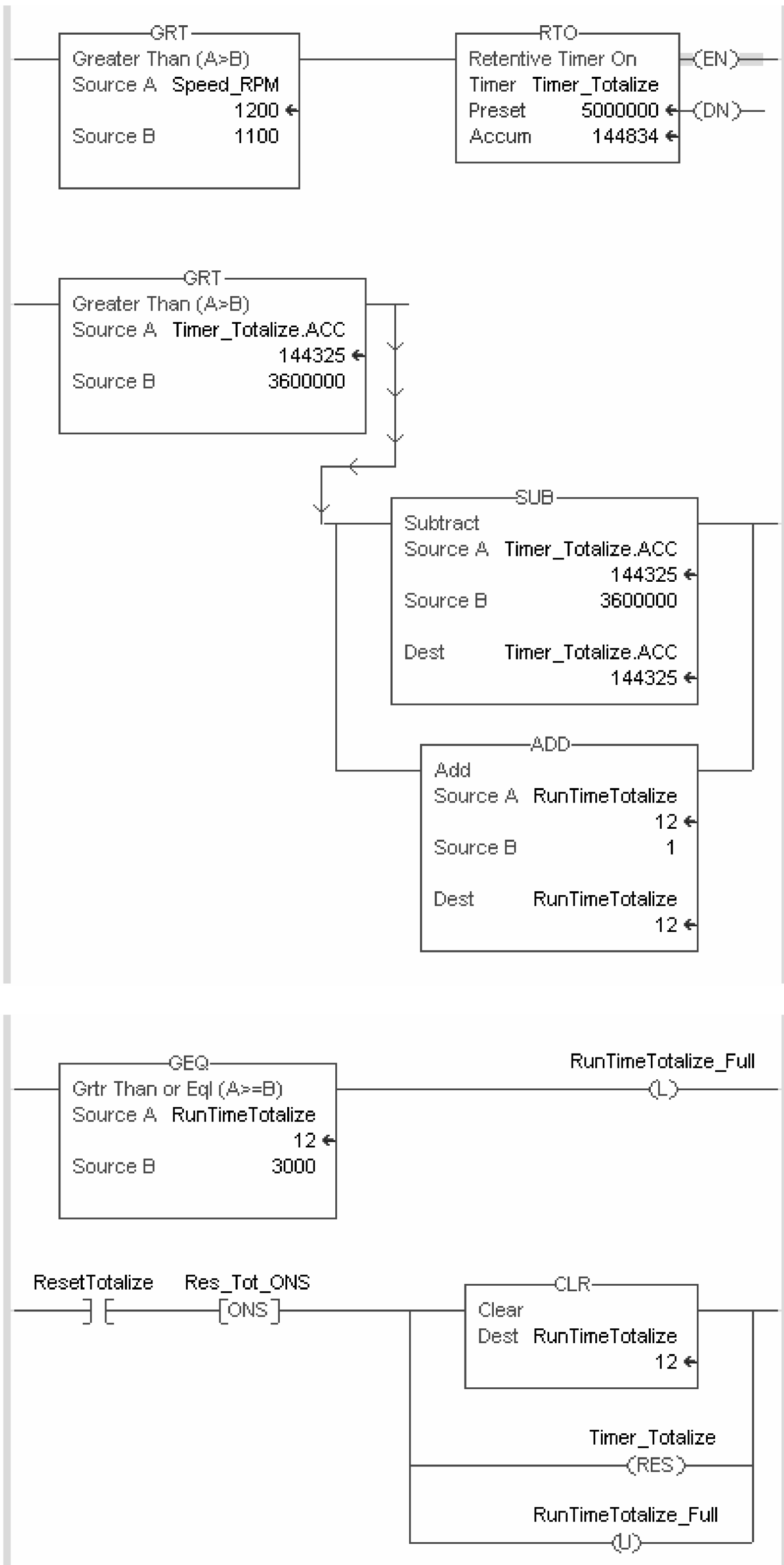


图 6-13　发动机运行时间积累完整的梯级逻辑

6.3　三角函数运算指令的编程

Logix 控制器的指令系统还可以完成比较复杂的运算，如指数对数运算和三角函数运算，指数对数的运算难以找到实例，本书暂不做讨论。三角函数运算指令如下：

- SIN　正弦指令
- COS　余弦指令
- TAN　正切指令
- ASN　反正弦指令
- ACS　反余弦指令
- ATN　反正切指令

作为时序逻辑控制应用为主的工控产品，三角函数的应用还真的不多见，这里，我们仅用一个正弦函数运算做一个尝试。有时编程产生一个正弦波用于测试，如功能块的高通和低通的模拟测试，改变正弦波的频率，可以观察滤波功能作用的结果，因而大大节省了现场调试的时间。

编写梯级逻辑如图 6-14 所示，这个例程被安排在一个周期任务中，设定每 2ms 执行一次，第一个梯级的计数器将按照每 2ms 加 1 的时间间隔计数，预置值设定为 360，每当完成位置位便复位该计数器。可以看出，这个计数器提供了一个从 0 ~ 360 的周期变化的均匀增长的数据，用此模仿角度的 0 ~ 360°。

第二个梯级是一个除法指令，将计数器提供的数据分割成若干等分，譬如除以 1、2、3、4 等，从而改变正弦波的导通角为 360°、180°、120°、90°等。除法指令执行的结果被送到目标地址的周期值标签 Cycles，该值将作为正弦波函数的运算参数。

第三个梯级将标签 Cycles 的值从角度转为弧度，采用的是转换指令 RAD，这条指令的执行结果是将角度的值转为弧度的值。

最后一个梯级执行正弦函数运算指令 SIN，运行结果将产生按照正弦波曲线变化的数据，可用 Trend 面板来观察曲线的变化。

现在，我们通过 Trend 面板观察当 DIV 除以 1 时，也就是 360°的导通角，正弦波的波形图如图 6-15 所示。这是一个完整的正弦波曲线。

当 DIV 指令除以 2 时，也就是 180°的导通角，正弦波的波形图如图 6-16 所示。这是一个半波曲线。

当 DIV 指令除以 3 时，也就是 120°的导通角，正弦波的波形图如图 6-17 所示。这是一个 2/3 个半波的曲线。

当 DIV 指令除以 4 时，也就是 90°的导通角，正弦波的波形图如图 6-18 所示。这是一个 1/2 个半波的曲线。

采用趋势 Trend 的观察面板来查看不同导通角下的正弦波形，由于是采用除法来改变同一周期时间下的导通角，我们看到的不同形状的波形都是同一个频率，这个频率就是计数器计数到 360 的周期时间。改变计数器的周期时间，正弦波的频率也就发生了改变，譬如，我们把计数器的计数脉冲分频一下，计数到 360 所需要的时间翻了一倍，如图 6-19 所示修改的梯级逻辑。

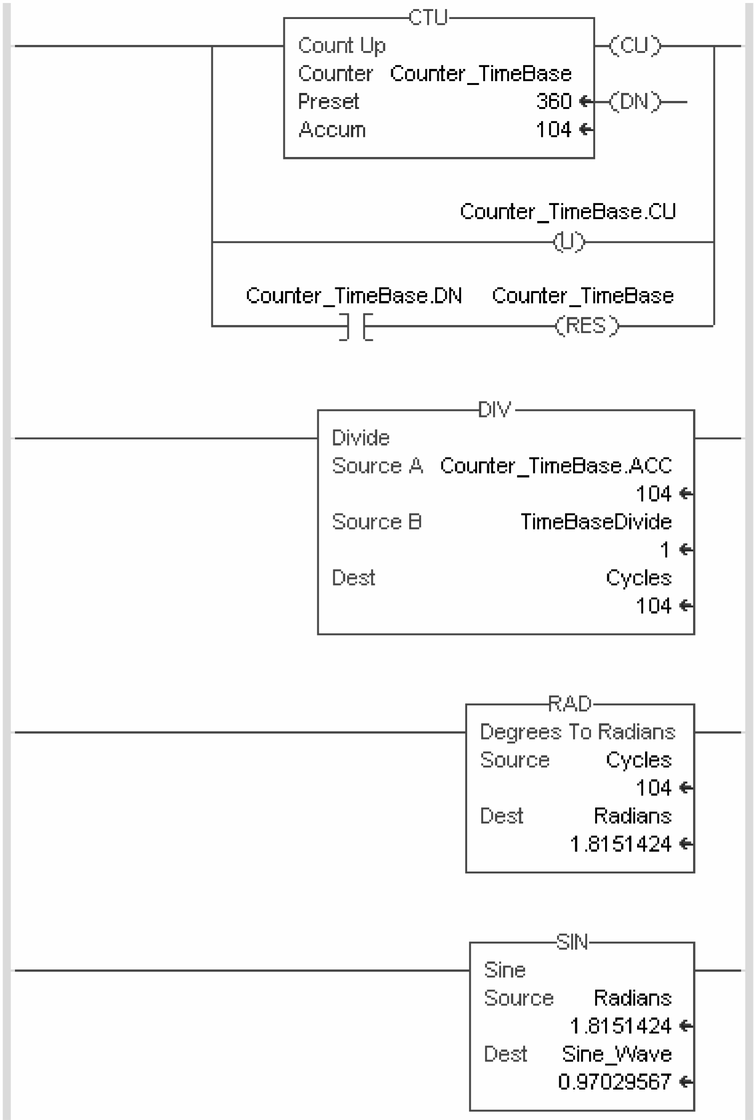


图 6-14 获得正弦波曲线的梯级逻辑

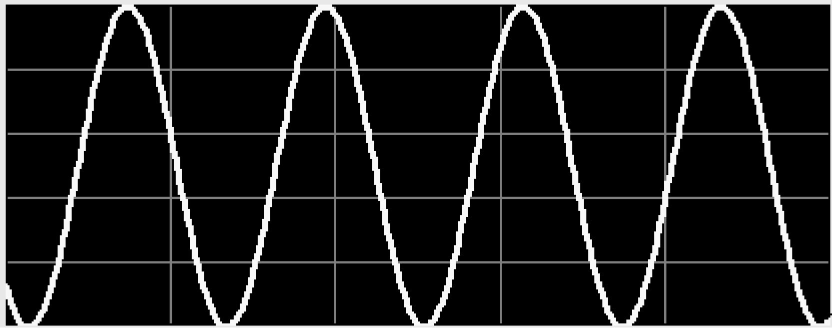


图 6-15 360°导通角的正弦波波形图

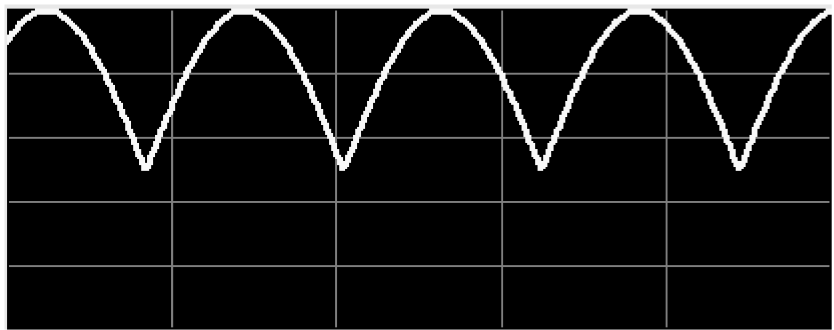


图 6-16　180°导通角的正弦波波形图

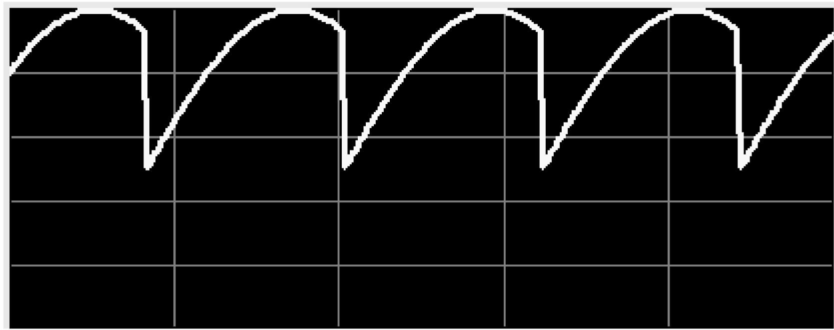


图 6-17　120°导通角的正弦波波形图

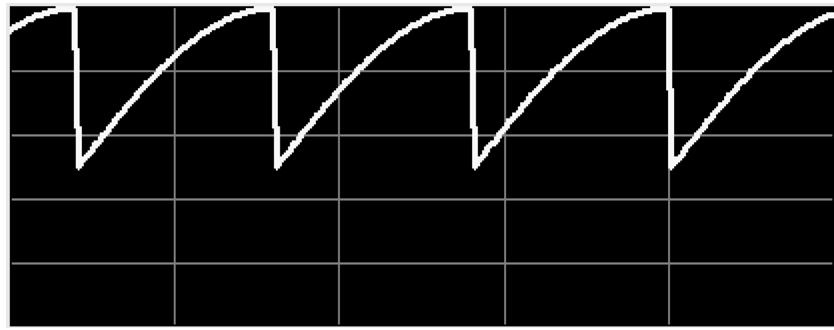


图 6-18　90°导通角的正弦波波形图

然后我们再用趋势 Trend 的面板来观察，波形图如图 6-20 所示，频率果然慢了一半。除了用分频计数改变计数器周期时间，还有别的方法吗？改变周期任务的调用时间，也可以收到相同的效果。现在我们周期任务的调用时间是 2ms，也就是每隔 2ms，计数器累加一次，如果改用 3ms 或者 4ms，波形频率将改变为原来的 2/3 或 1/2。总之，改变计数器的计数频率就改变了正弦波的频率。

显然，正弦波指令运算的结果，幅值是原始的，将标签 Sine_Wave 乘以某个系数便可得到某个幅值的正弦波。

现在我们再来看看正弦波运算的另外一种用法，梯级逻辑编写如图 6-21 所示。这是以产生正弦波为例的一段梯级，假定时间周期设置值是以秒为单位的，通过乘法指令，换算为毫秒为单位的计时器所需要的预置值，一个自复位计时器所提供的重复变化的累加量数值，为正弦波值计算提供一个变化参数。

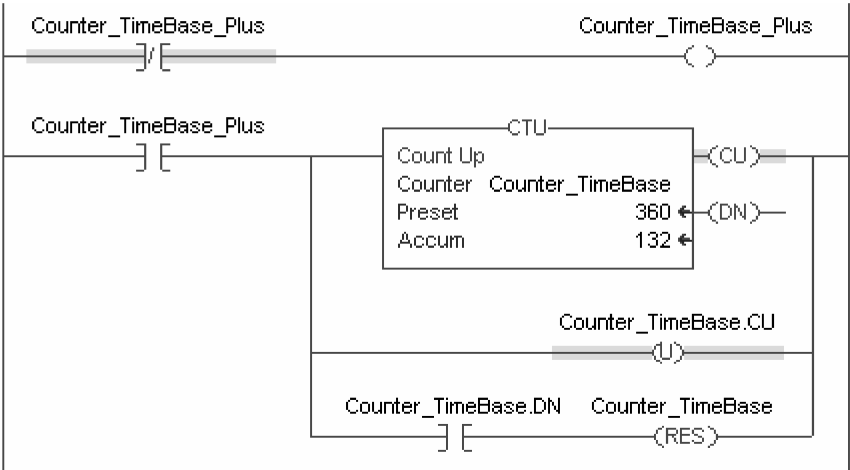


图 6-19　分频修改正弦波频率的梯级逻辑

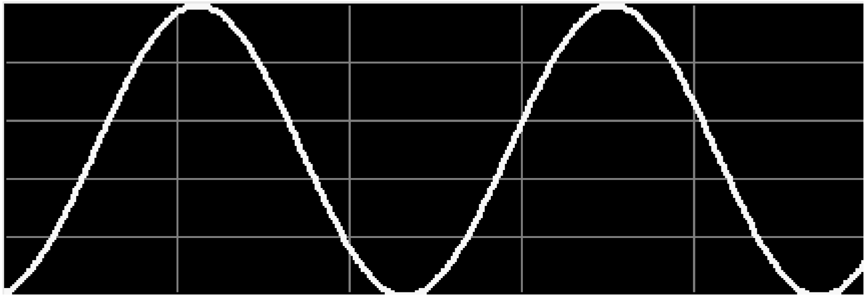


图 6-20　改变频率后的正弦波波形图

最后一个梯级的 CPT 指令的计算公式，计算结果为正弦波，注意单一运算指令 SIN 被 CPT 指令的计算公式引用，完成正弦波运算。

计时器预置值用作正弦波计算的比值分母 XDen，计时器累加值用作正弦波计算的变化比值分子 XNum。这两个参数的比值，提供了一个在给定周期时间内的 0 – 1 的变化量，由于这个比值分子 XNum 的变化量是与扫描周期时间有关的 0 ~ 5000ms 的精细时间的积累，0 – 1 的实数变化也十分精细，这个精细数据通过 SIN 指令的运算，得出一个精细的正弦波曲线。标签 Max 为一个实数，用作正弦波的最大幅值，修改这个值就可以改变正弦波的幅值。目前，这个值在数据库标签的数据表中被设置为 10。

创建一个趋势 Trend 面板，用来监视数据 Out_Sinewave 标签，可观察到正弦波形，如图 6-22 所示。这是一个每 5s 一个完整正弦波的频率，趋势 Trend 面板设置的 X 轴的时间是 20s，正好可以看到 4 个完整的正弦波。

将周期时间标签 Period 从 5s 改为 8s，即正弦波的频率改变为 8，趋势 Trend 面板波形如图 6-23 所示，可以观察到改变了正弦波频率的波形，X 轴的 20s 时间能看到的是两个半正弦波波形。

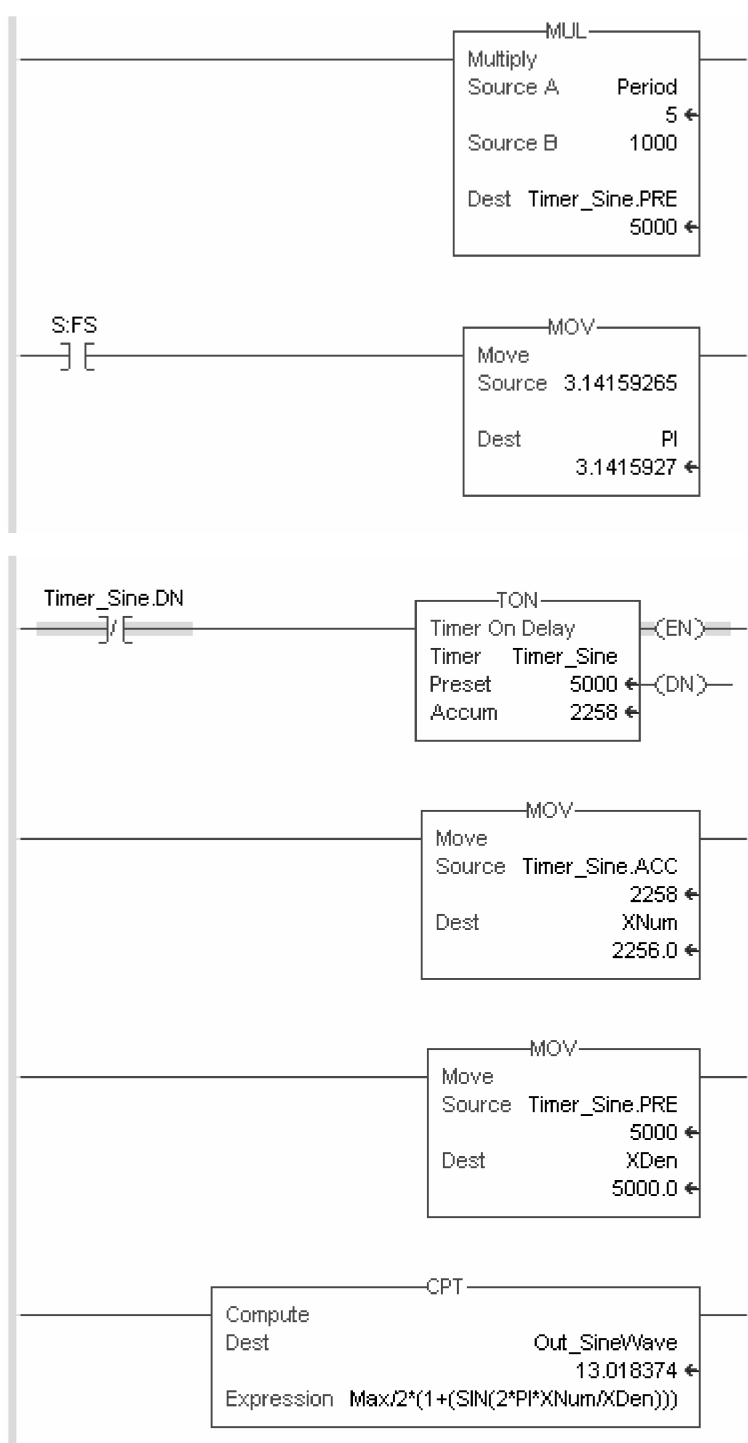


图 6-21　产生正弦波波形的梯级逻辑

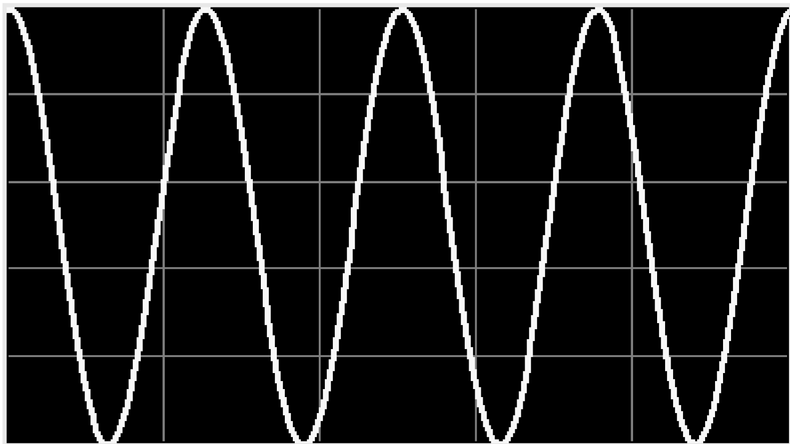


图 6-22 频率为 5s 的正弦波波形

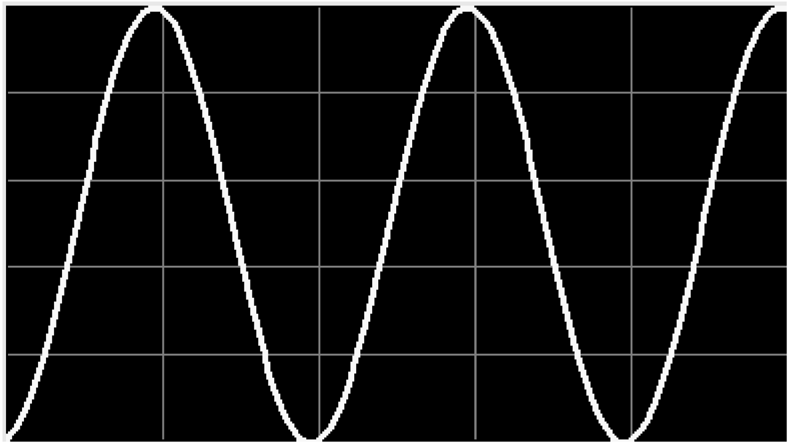


图 6-23 频率为 8s 的正弦波波形

将 Max 标签的数值在数据表中从 10 改为 8，趋势 Trend 面板波形如图 6-24 所示，可见改变了正弦波的幅值，可观察到波形的幅度降低到了原来的 4/5。

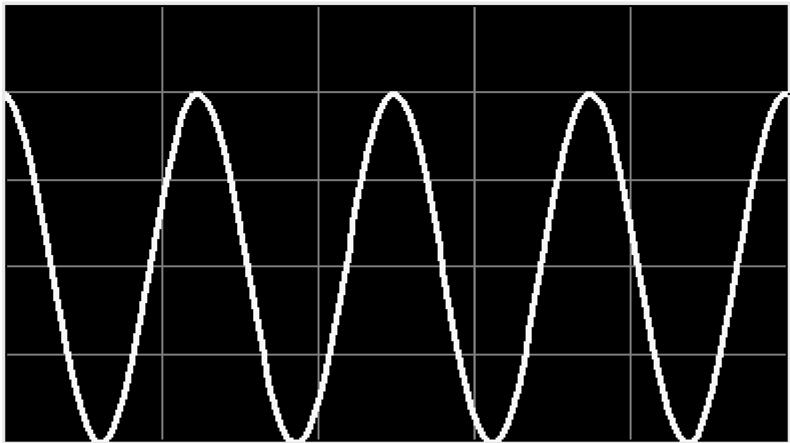


图 6-24 频率为 5s 降幅至 4/5 的正弦波波形

能够观察到一个完美的正弦波跟好几个因素有关：跟例程的扫描时间有关，跟趋势 Trend 采样的时间有关，跟趋势 Trend 的 X 轴设置即时间坐标有关，跟 Y 轴设置即幅值坐标有关。例程足够快的扫描时间，计时器的累积值和正弦波形值的计算才会细腻；趋势 Trend 采样时间最好跟例程扫描时间基本吻合才能充分表现线条的圆润；X 轴是时间轴，适当的选择可以观察稀疏适度的波形，这自然跟周期有关联，此处 X 轴设置的是 20s，即数值轨迹标点出现在屏幕直到消失需要 20s 的时间；Y 轴可以选择略高过幅值的坐标值，也可以选择自动适应幅值的方式。

6.4　逻辑运算指令的编程

逻辑运算指令是字操作指令，对一个字中的每个位进行逻辑操作，可用来完成位的批量处理，也是传统控制器产品常用的编程手段。逻辑运算指令的执行，基本上是寄存器的运算方式，留有很多硬件模式的处理痕迹。当一个硬件控制系统升级到可编程控制系统时，很多相应的硬件逻辑门的处理就转变成了逻辑运算处理。

逻辑运算指令如下：

- AND　与逻辑运算指令；
- OR　或逻辑运算指令；
- XOR　异或逻辑运算指令；
- NOT　求反逻辑运算指令。

逻辑运算是针对字来操作的，但我们需要查看的却是位，可在数据表中，将操作对象的标签从十进制显示方式改为二进制显示方式，这样可以得到离散量的表达形式，看起来更直观。下面梯形图中的逻辑运算对象的标签在数据表中是改成了二进制显示方式的。

逻辑运算在实际运用中比较常见的是位的异或运算，利用这个运算关系可以判断出位状态是否发生变化，这正是很多运行状况所需要的判断，编写梯级逻辑如图 6-25 所示。

当梯级条件成立时，输入字 Input1 与参考字 Refrence1 进行异或运算，输入字 Input2 与参考字 Refrence2 进行异或运算，每个字的所有位同时参加异或运算，相同结果为 0，不同结果为 1。只要结果字 Result1_XOR 或 Result2_XOR 中的任何一个位是不同结果为 1，也即字中的任何一位状态发生了改变，则比较指令的梯级条件成立，将找到位 Find1 或 Find2 置位并锁定。

根据产生的结果不同，Find1 或 Find2 所控制的后续梯级逻辑还要作相应处理。此处使用大于指令 GRT 可判断字的结果是非 0 的情况；使用不等于指令 ENQ 也可判断字的结果是非 0 的情况。

异或的运算属于字操作指令，一次只能处理一个字，当我们需要很多个字进行异或运算比较时，不得不重复地进行如上操作，耗用了大量梯级逻辑，这在早期梯形图逻辑中是很常见的。在后面的第 12 章特殊指令中，我们将看到异或操作数组处理的威力，不过相应的那条指令并不直接称为异或数组运算，但它所完成的任务却实实在在是异或数据的批量操作。

在第 12 章特殊指令中，我们将讨论数组操作的诊断检测指令 DDT 的批量数据比较结果的处理方法，其比较结果的记录及时而完整。

还有一种处理需求是，不但需要判断比较对象的不同，还要改写参考量，令参考量与刚

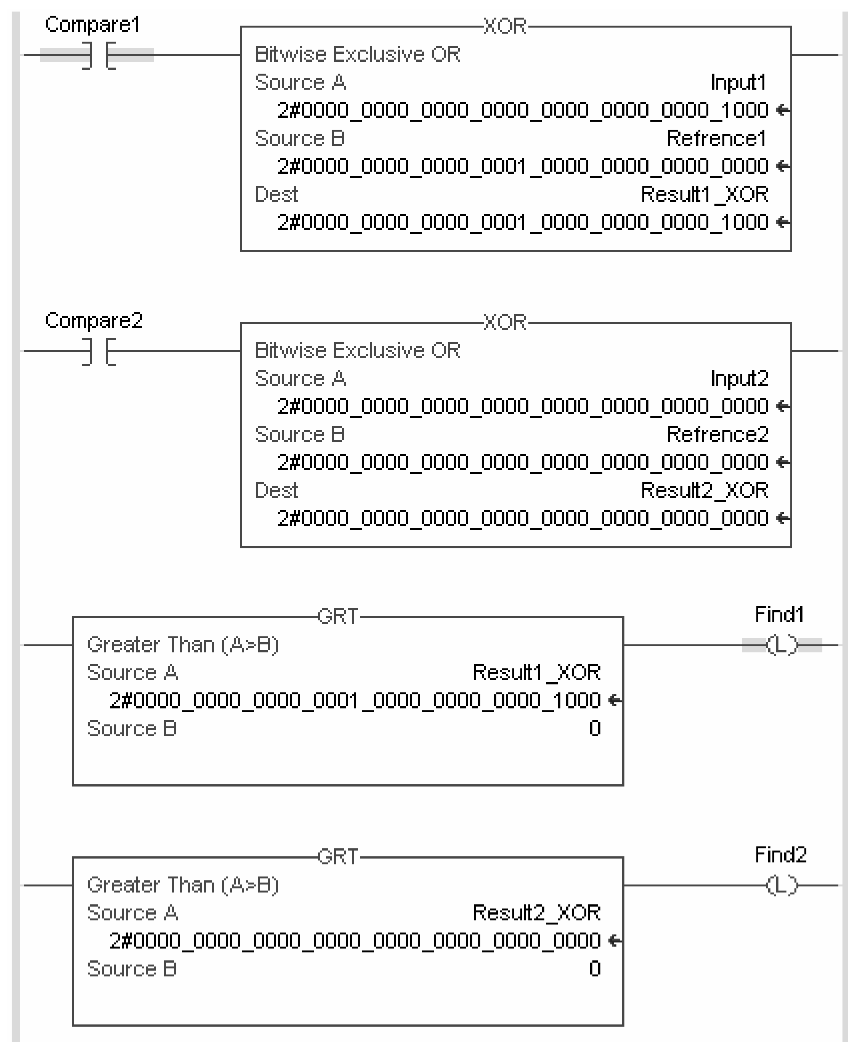


图 6-25 异或比较梯级逻辑

发生改变的比较对象一致，等待下一次的比较，如图 6-26 所示的梯级逻辑。

当梯级条件成立时，输入字 Input3 与参考字 Referce3 进行异或运算，输入字 Input4 与参考字 Refrence4 进行异或运算，每个字的所有位同时参加异或运算，相同结果为 0，不同结果为 1。只要结果字 Result3_XOR 或 Result4_XOR 中的任何一个位是不同结果为 1，也即字中的任何一位状态发生了改变，则比较指令的梯级条件成立，将找到位 Find1 或 Find2 置位并锁定。然后将输入字传送到参考字，修改后的参考字将作为下次的比较参考。

找到位 Find3 或 Find4 作为某个梯级条件，用来控制因发生变化而要执行的动作。

完成这种需求的数组处理指令是 DTR，这是一条输入指令，用来监视数据的变化，一旦数据发生变化，将产生梯级条件为真，用来控制后面的输出指令。DTR 指令将在后面第 12 章特殊指令中讨论。

Compare3

XOR
Bitwise Exclusive OR
Source A Input3
2#0000_0000_0100_0000_0100_0000_0010_0000
Source B Refrence3
2#0000_0000_0100_0000_0100_0000_0010_0000
Dest Result3_XOR
2#0000_0000_0000_0000_0000_0000_0000_0000

Compare4

XOR
Bitwise Exclusive OR
Source A Input4
2#0000_0000_0000_0100_0000_0100_0000_0000
Source B Refrence4
2#0000_0000_0000_0100_0000_0100_0000_0000
Dest Result4_XOR
2#0000_0000_0000_0000_0000_0000_0000_0000

GRT
Greater Than (A>B)
Source A Result3_XOR
2#0000_0000_0000_0000_0000_0000_0000_0000
Source B 0

Find3
(L)

MOV
Move
Source Input3
2#0000_0000_0100_0000_0100_0000_0010_0000
Dest Refrence3
2#0000_0000_0100_0000_0100_0000_0010_0000

GRT
Greater Than (A>B)
Source A Result4_XOR
2#0000_0000_0000_0000_0000_0000_0000_0000
Source B 0

Find4
(L)

MOV
Move
Source Input4
2#0000_0000_0000_0100_0000_0100_0000_0000
Dest Refrence4
2#0000_0000_0000_0100_0000_0100_0000_0000

图 6-26　异或比较参数修改梯级逻辑

从以上两例逻辑运算来看，异或运算只是单一的运算，通过这样的逻辑运算得到一个结果，这很像硬件寄存器的处理方式，要有更多的判断和处理，还需要配合其他梯级的逻辑编写。异或运算一旦进化成数组处理方式，不但可以进行批量的处理，还具有判断和记录的功能，在后面的章节我们会继续讨论。

在任何算术逻辑运算中，不管是源字还是目标字，使用短整型字或整型字是很不合算的，因为这样的数据类型进入 CPU 处理之前要换成 CPU 能处理的基本数据单元，处理之后的数据仍为基本数据单元，还要换为指定的短整型字或整型字，这个过程不但需要耗用额外的数据单元来缓存数据，还要耗用时间来转换，这种时间空间的消耗可能超乎了你的想象，比起使用基本数据单元操作数，内存耗用和时间消耗竟然多达 10 倍以上，这就是我们在很多手册中看到反复强调要使用双整型字和实数字的原因。在编程软件的指令集介绍中，可以看到大多数指令可使用的数据类型列表中，双整型字和实数字是用黑体字显示的，表示推荐使用的数据类型，请留意。

所有涉及字以及字数组的指令操作都是同样的道理，有关字操作的指令，源地址和目标地址的数据类型选择必须基于这样的考虑。千万不要以为使用更短的数据类型可以节省空间，这是极大的误解。当我们不得不使用单整型字时，大多是因为需要跟传统的处理器 PLC5 或 SLC500 进行数据的交换，一般情况下应该尽可能地避免使用非基本数据单元的数据处理，尽可能地使用双整型字和实数字。

第 7 章

比较指令的编程

比较指令是典型的输入指令，梯级条件的判断除了位逻辑运算的结果，使用最多的当推比较指令了。目前的控制器具有充裕的内存空间和高速的 CPU 运行，无须考虑节约内存和受限于运行时间，所以梯级条件使用比较指令的做法越来越普遍，这是一种表达清晰的判定关系，容易辨识和解读。比较指令是针对数轴的一段范围进行判定，以不等式的成立与否来决定梯级条件。在 Logix 控制器的指令系统中，有一套比较指令来表示不同的比较关系：

- EQU 等于比较指令
- NEQ 不等于比较指令
- LES 小于比较指令
- LEQ 小于等于比较指令
- GRT 大于比较指令
- GEQ 大于等于比较指令
- CMP 表达式比较指令
- LIM 数据范围比较指令

7.1 等于指令 EQU 和不等于指令 NEQ 的编程

前面章节我们曾经讨论过，在实际运用中，常常会有指令执行动作的互锁，固然指令动作的互锁是常用的方法，当多条指令执行需要互锁时，指令的使能位互锁便显得比较繁琐。在这种情况下，建议采用指针的方法来解决执行的顺序和确保执行动作的惟一性，作为识别判定的条件，更为直观和简洁，这时等于指令 EQU 的运用就必不可少了。

例如，一个数据长度有限的对外通信数据缓冲区，假定一次只能传送 100 个双整字，现在有 500 个双整字需要传送，只能分为 5 次轮流传送，这些传送的执行动作必须互锁，既要有一定的传送顺序，又要确保任何时候只有一段数据正在传送，可以安排长度为 5 的指针来协调工作，编写梯级逻辑如图 7-1 所示。

定义对外的通信缓冲区为数组标签 Array_Carry，这个标签的数据是动态的，按顺序地从传输对象数组标签 Array_Transmission 中截取一段数据块装入这个缓冲区，数据传送装载工作交给 COP 指令完成，完成的动作顺序由指针来安排，等于指令 EQU 区别不同的梯级条件，选择不同的数据段，COP 传送到作为数据缓冲区的数组标签 Array_Carry 中。

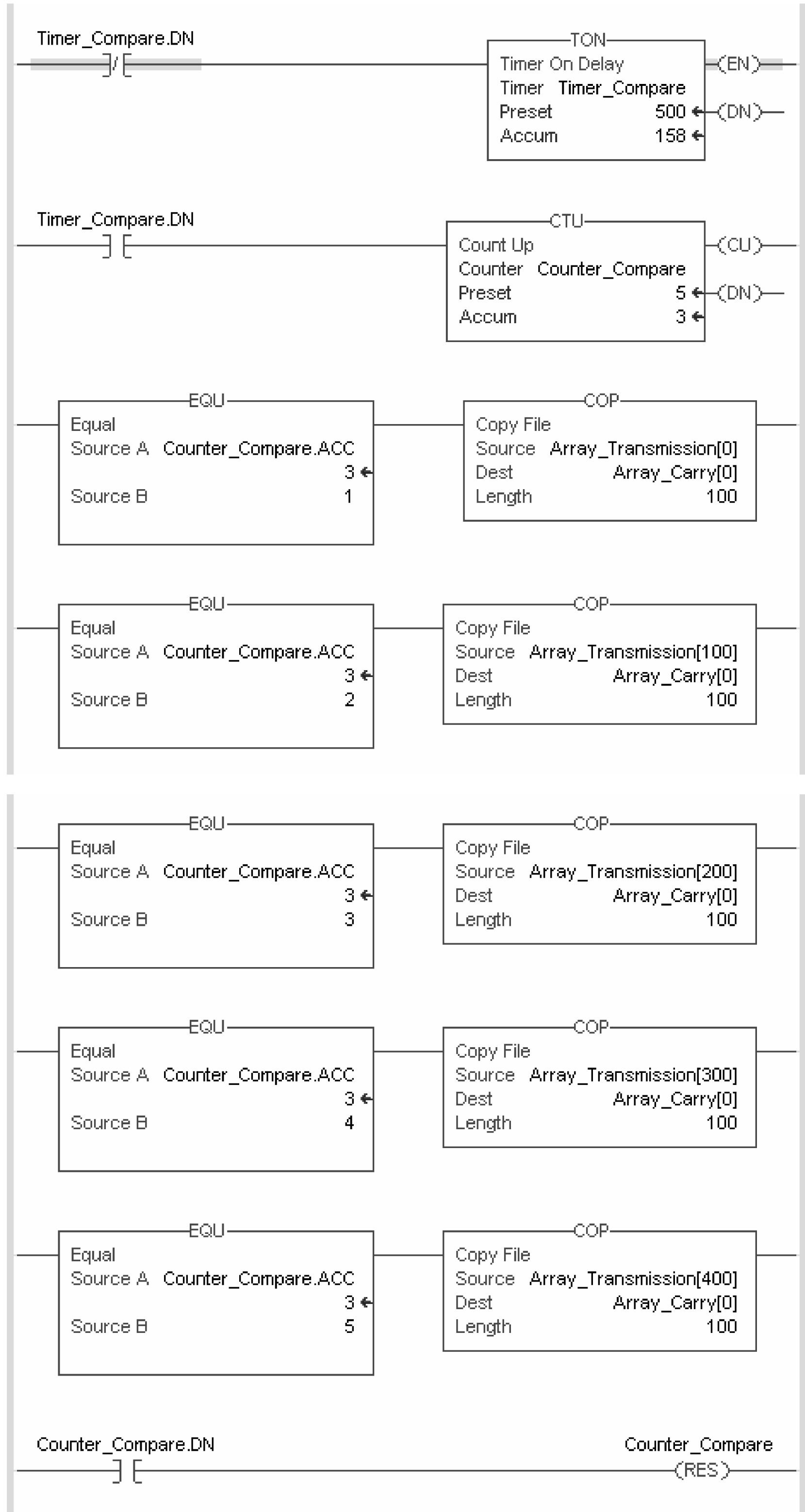


图 7-1　指针控制传输顺序的梯形逻辑

自复位计时器 Timer_Compare 提供计数脉冲，作为指针的计数器按照它的时间间隔计数。每当计数器计数到 5 时，完成位置位，用来复位计数器 Counter_Carry，从而一个数据传送的周期完成，即 500 个双整字全部传送一次。根据需求，数据传送的频率可以通过改变计时器的预置值来设定，此例中是 0.5s 传送一个数据块，所有数据全部传送一遍需要 2.5s 的时间。

请注意，采用计数器完成位执行的计数器复位动作，宜放在所有 COP 指令完成之后的梯级进行，也即跟计数器累加值有关的执行动作全部执行完毕，再复位计数器。

试想一下，如果将计数器复位的梯级紧接着放在计数器指令计数的梯级之后会怎么样呢？我有时读到的梯形图程序，EQU 指令判定对象的指针是 0、1、2、3、4，没有选择直接了当的 1 ~5 的表达，这个编程者一定经历了一个这样的过程，始终得不到等于 5 的判断条件，于是退而求其次，选择了 0 ~4 的判断，这也何尝不可，照样按 5 个步骤执行。得不到等于 5 的判断条件的原因是计数器复位的动作提早了，没有在所有的动作执行完毕再复位，所以，请不要抱怨计数器指令不好使，可能是你对指令使用的细节研究不够。有一次，我读到一段例程的结尾，如图 7-2 所示，他应该是把这个例程中所有的复位动作统统放在例程的最后扫描来处理，绝无疏漏且一目了然。还是那句话，良好的编程习惯可以帮助你避开陷阱。

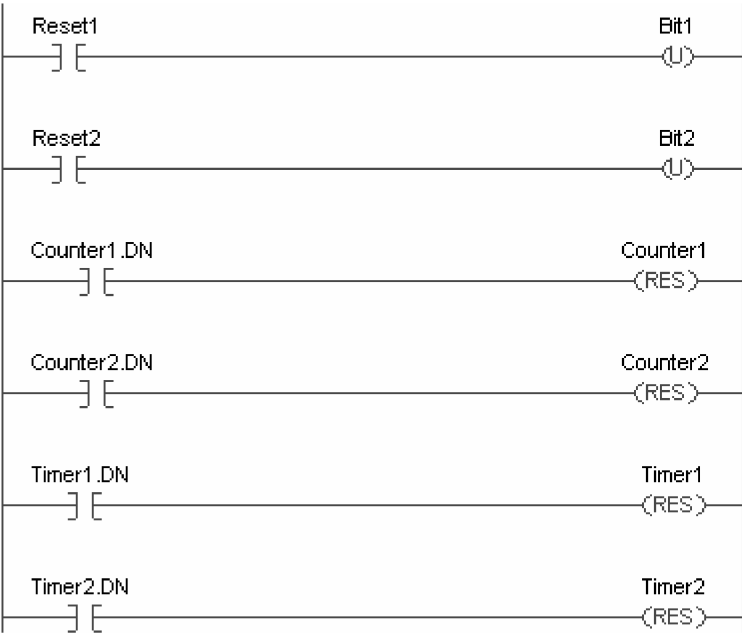


图 7-2　一段例程的结尾

在计数器的章节里，我们曾用一个计数器产生的不断变化的数据来作为一个主控制器发出的脉搏信息，表示主控制器处在运行状态，那么在每个从控制器中，应该对这个收到的来自主控制器的脉搏信息进行判断，让我们编写梯级逻辑来实现，如图 7-3 所示。

每当梯级被扫描时，不等于比较指令 NEQ 将判断运行测试标签 Run_Test 和它的缓冲标

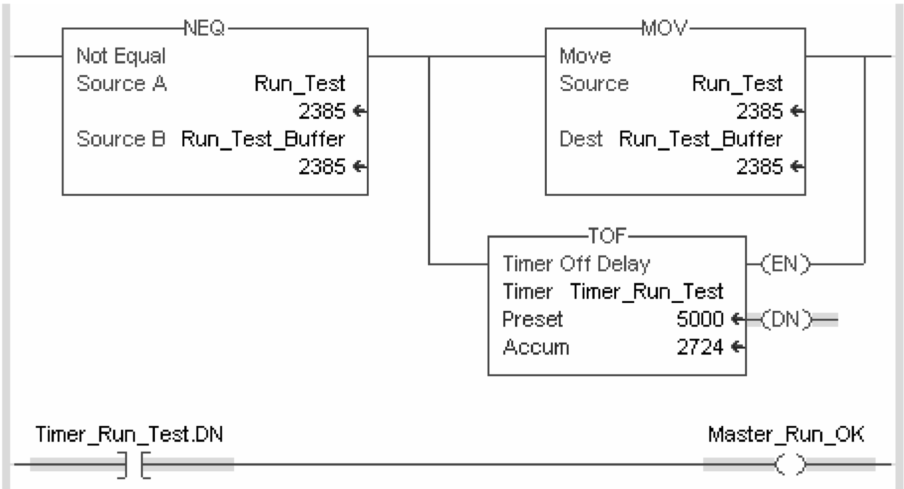


图 7-3　运行判定梯级逻辑

签 Run_Test_Buffer，当它们不相等，梯级条件成立，MOV 指令则将本次的数据传送到缓冲标签，准备下一次的比较，此时表示对方主控制器正处在运行的状态，它的数据正在变化。断延时计时器 Timer_Run_Test 在数据未刷新期间连续计时，数据刷新后则复位计时器，只要未达到预置值，它的 DN 位状态始终为 1，维持着表示运行状态正常的输出标签 Master_Run_OK。一旦比较结果相等的时间超过预期的值，说明数据一直没有更新，此时梯级条件不成立，MOV 不再传送数据，当断延时计时器的累积时间达到了预期值，这个预置值为 5s 的计时器，在延时 5s 后，DN 位复零，表示运行状态正常的位 Master_Run_OK 被关闭。你应该还没有忘记吧，当断延时计时器的累加值 ACC 达到预置值 PRE 的时候，它的 DN 是复零的状态。

7.2　综合比较指令 CMP 的编程

CMP 指令是一条综合比较指令，其优点是可以综合运算表达式，一目了然地集中了运算关系和比较关系，这是常常使用的一条指令。

例如，某个需求是欲获得一个闪烁的灯，并且这个灯的闪烁频率是可修改的，编制梯级逻辑如图 7-4 所示。

一条 MOV 指令修改闪烁的频率，也即计时器的预置值，一条自复位的计时器指令提供了闪烁的周期，因为采用了表达式的比较，修改计时器的预置值就很容易改变灯的闪烁频率了。这里表达式一半的时间梯级条件成立令灯亮，一半的时间梯级条件不成立令灯灭。这是亮灭相等时间的闪烁，如果想要得到不同的亮的时间和灭的时间，修改 CMP 指令表达式中计时器预置值的除数就好了，比如想要 1/4 的时间灭，3/4 的时间亮，用 Timer_Flash. PRE/4 的表达式就可以了。当然，换成小于比较也是可以做的。想想看，怎样改变表达式。

人的视觉残留时间为 200ms，决定闪烁的频率的计时器预置值设定为 400ms，即 200ms 的时段亮；200ms 的时段灭，这正好满足人的视觉残留时间参数，使人感觉到灯的闪烁，频率太慢难以感觉到闪烁；频率太快只感觉灯变暗而没有亮灭的节奏。

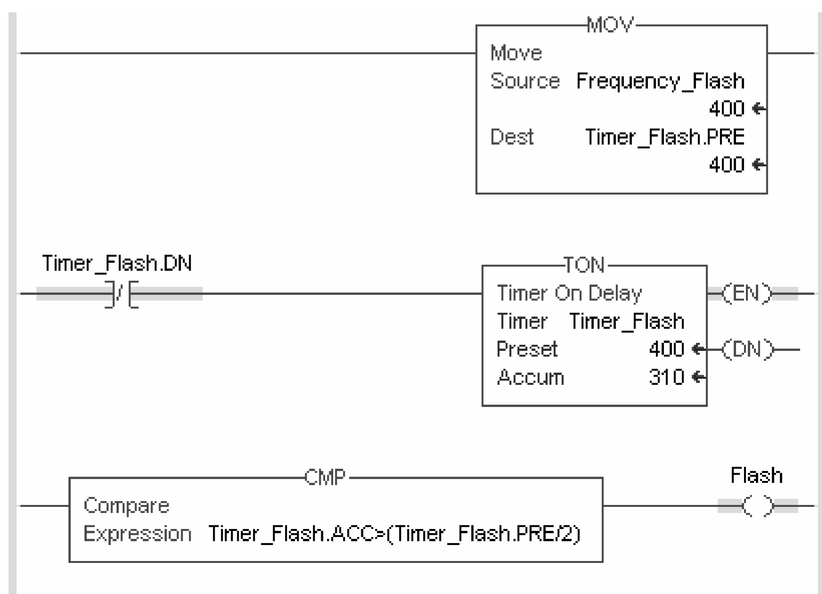


图 7-4　编制的梯级逻辑

此外，如果单一的比较关系，则不宜采用较耗费时间的 CMP 指令，建议用单一比较指令，比如大于的比较就直接用 GRT 比较指令，而不要用 CMP 的表达式来比较，尽管 CMP 指令用“ >”的比较符也是可行的。

7.3　范围比较指令 LIM 的编程

LIM 指令是一段范围值的比较，有趣的是，将比较临界点的指令参数低限值 Low Limit 和高限值 High Limit 置于数轴不同位置，会产生不同的比较范围。如图 7-5 所示，低限值 Low Limit 小于高限值 High Limit，所确定的是范围之内。如图 7-6 所示，低限值 Low Limit 大于高限值 High Limit，所确定的是范围之外。

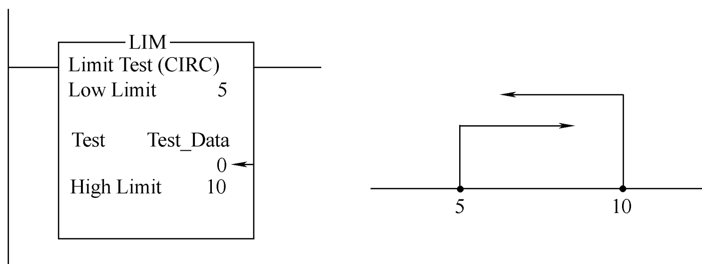


图 7-5　范围内的比较

LIM 指令的前身应该是两条比较指令的串联，如图 7-7 所示的 GEQ 和 LEQ 指令，等同于图 7-5 所示的范围比较关系；如图 7-8 所示的 GEQ 和 LEQ 指令，等同于图 7-6 所示的范围

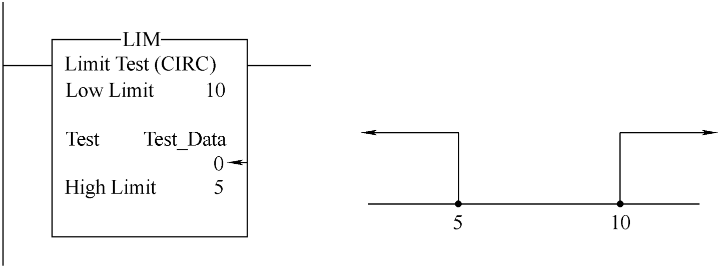


图 7-6　范围外的比较

比较关系。这种技巧性的范围比较关系的编程，遂转为 LIM 指令。这里我们又一次看到指令进化的实例。

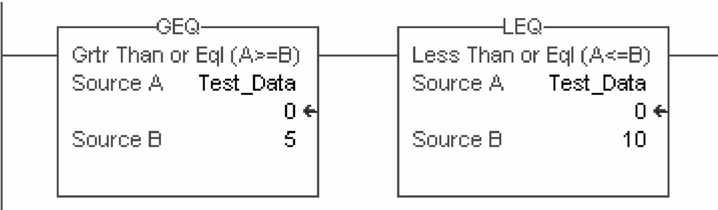


图 7-7　范围内的比较

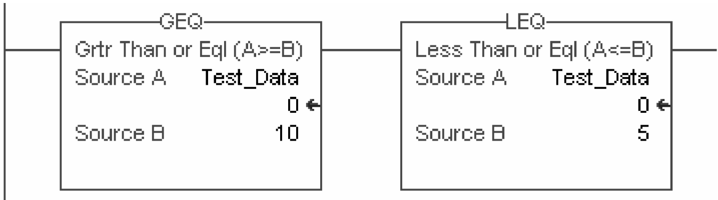


图 7-8　范围外的比较

有一个需求是，某个包装工艺中，以红、绿、蓝三种颜色的球来识别三种不同的包装过程，当红球弹出，光感传感器识别后送到控制器来的数据为 1200～1500；当绿球弹出，光感传感器识别后送到控制器来的数据为 800～1100；当蓝球弹出，光感传感器识别后送到控制器来的数据为 300～600。根据不同的判断结果，将调用不同顺序器输出指令数组来控制包装过程。我们试着用 LIM 指令来识别范围，并作为判断条件。

分析 3 个范围的分布，如图 7-9 所示，这是一个范围内的判断。

解读如图 7-10 所示的梯级逻辑，用 LIM 指令识别了 3 个范围的数据，从而产生梯级条件。显然这 3 个梯级条件绝不会同时存在，每个梯级的输出指令的实施锁定了自身的操作并解除其他两种操作，不管其他两种操作当时处在何种状态，都予以解除。三者必居其一的互

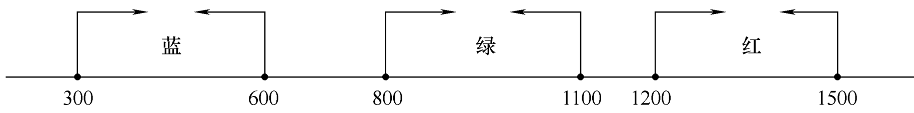


图 7-9　3 个范围的分布

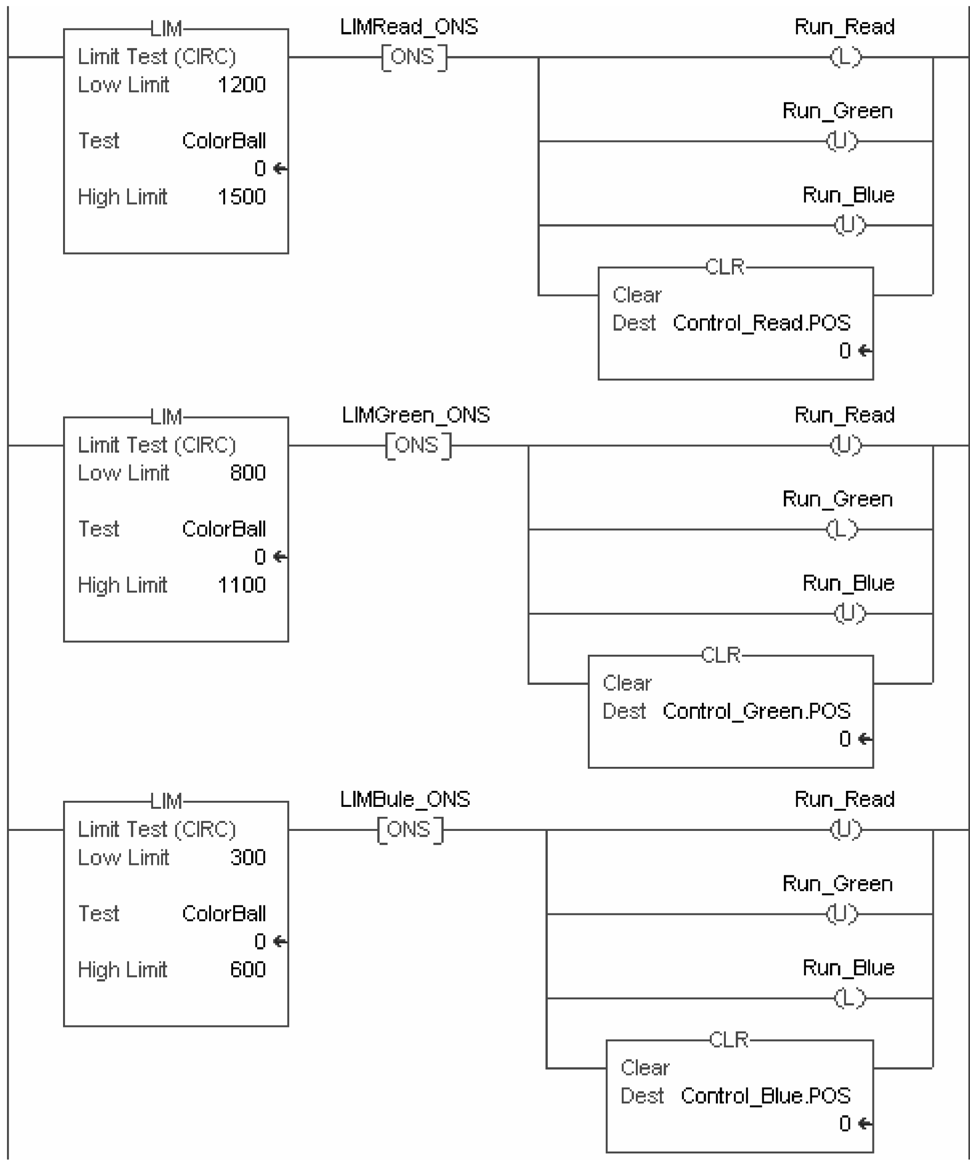


图 7-10　识别 3 个范围的梯级逻辑

锁关系，确保了只有一种操作方式存在，这是一种非常严密的模式选择方式，有经验的工程师通常会采用这样的梯级逻辑处理互锁关系，看似繁琐却极为可靠。

同时，还为每种操作的进入做了初始化的工作，将相应顺序器输出指令的指针清零。梯级条件之后配合 ONS 指令，确保了锁存解锁和清零的操作只会进行一次，凡是锁存解锁的操作，都无须多次操作，这是常规的配合关系。

前面的梯级锁定了操作，三种被相互锁定的操作，只有一个梯级条件是成立的，各自在条件满足时按其步骤进行，顺序步的进程则由 Step_Plus 来控制。顺序器输出的工作过程如图 7-11 所示，我们现在暂时不做深究，后面有专门的章节讨论。

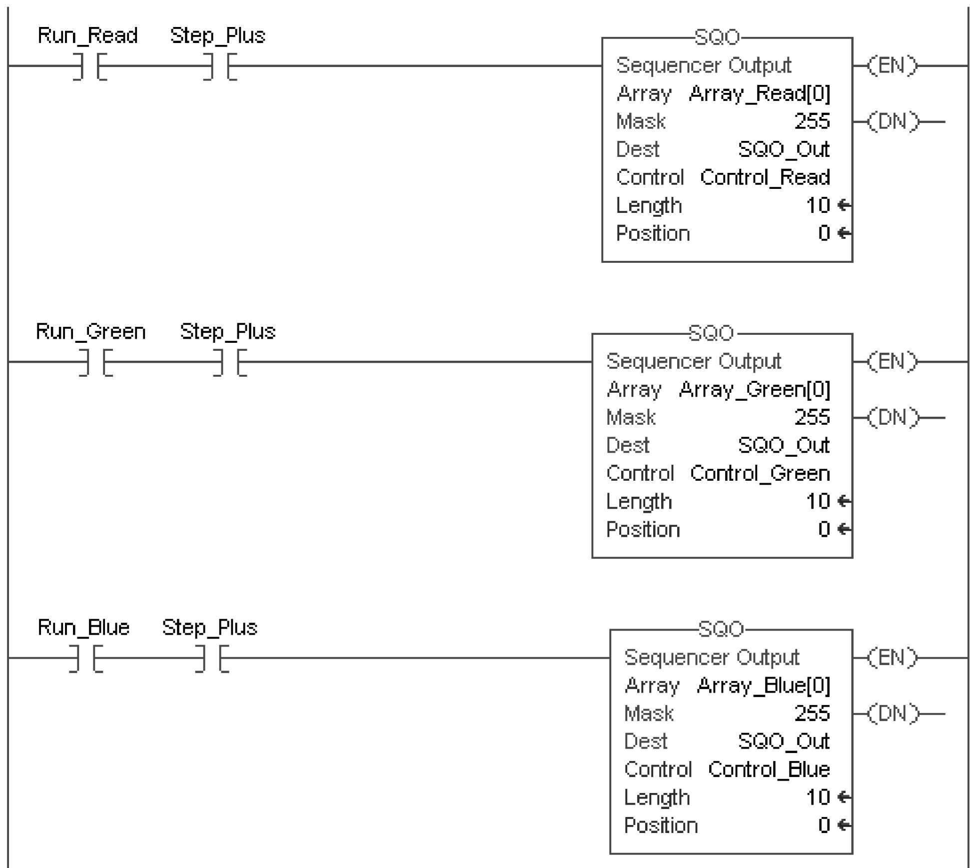


图 7-11　选择顺序器输出的梯级逻辑

下载到控制器运行，调试过程中你会发现，在数据范围判断过程中，还有数据空档部分，这些数据区域不会有新的控制结果产生，输出控制会残留在原来终止的状态，上一次的

操作还在锁定状态，这些都不属于三种操作之一，应该做相应的处理，比如说复位。编写梯级逻辑如图 7-12 所示。

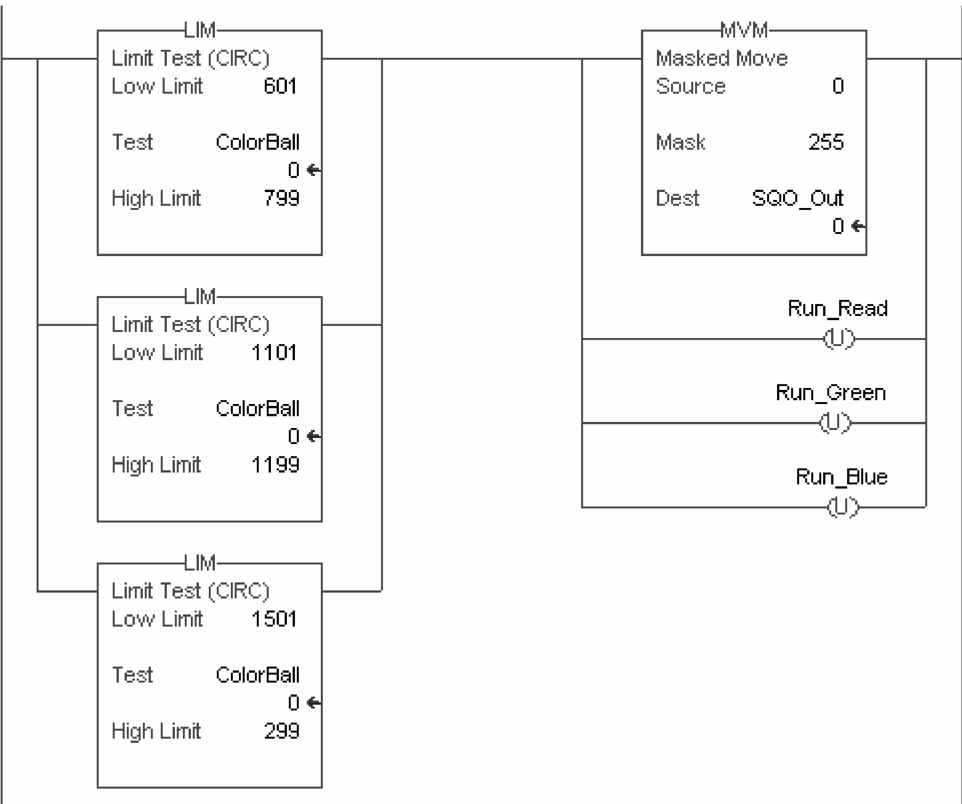


图 7-12　不在操作范围内的比较结果

这三条 LIM 指令的用法有些不同，前面两条是对范围内的确认，即 601～799 和 1101～1199；后面一条是对范围外的确认，即大于 1501 或小于 299，这 3 个比较条件的任一个满足，都属于非正常状态，都要执行清除处理的操作。所执行的带屏蔽传送指令，仅仅是将顺序器输出的低 8 位清除为零。

综合以上的梯级逻辑编写和执行的梯级逻辑顺序，完整的过程如图 7-13 所示。请注意梯级排列的前后顺序，即执行的顺序，这是任何时候都要强调的，执行顺序和我们编程时所思考的顺序有时是不同的。

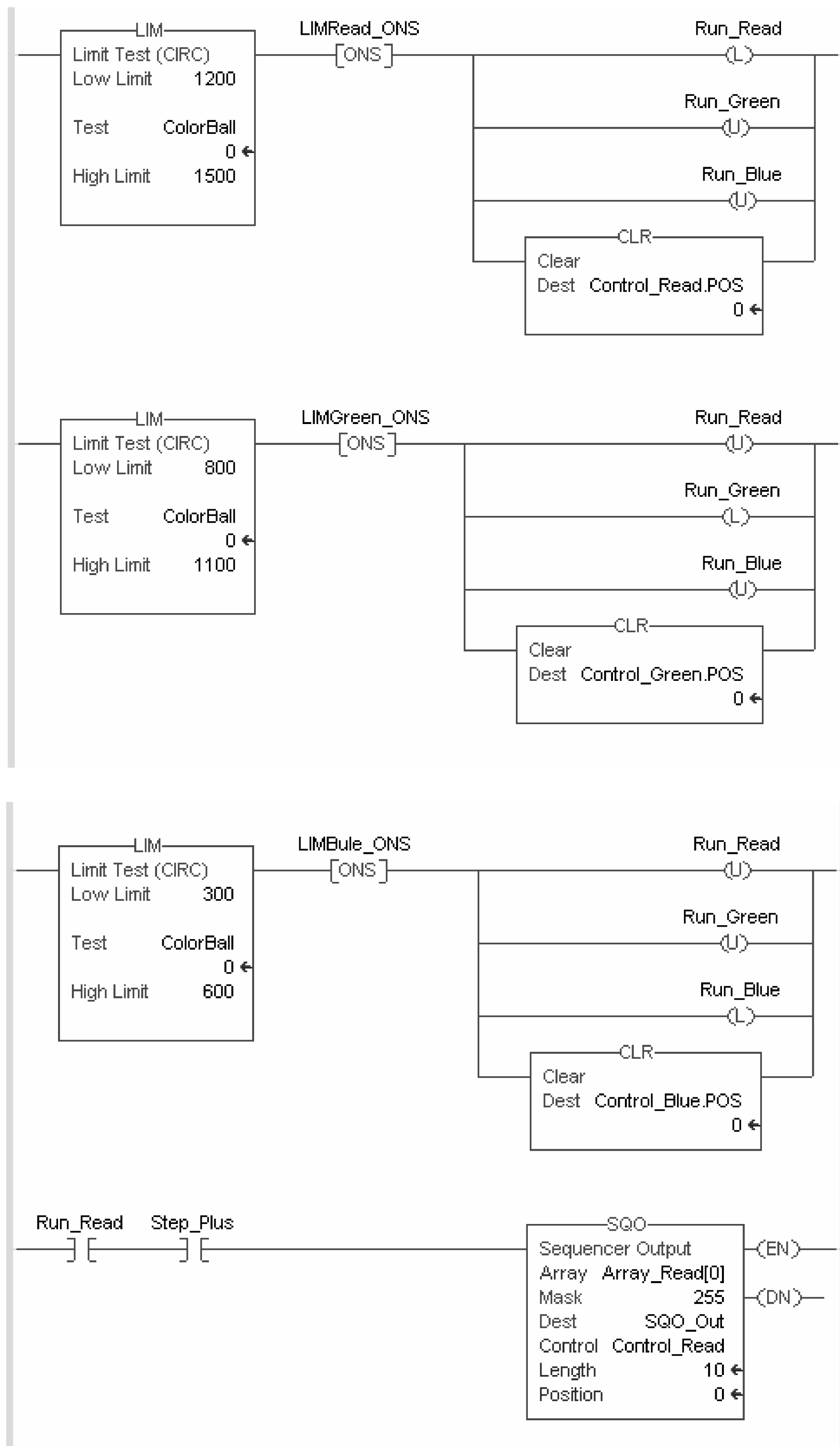


图 7-13　范围判断和执行的完整的梯级逻辑

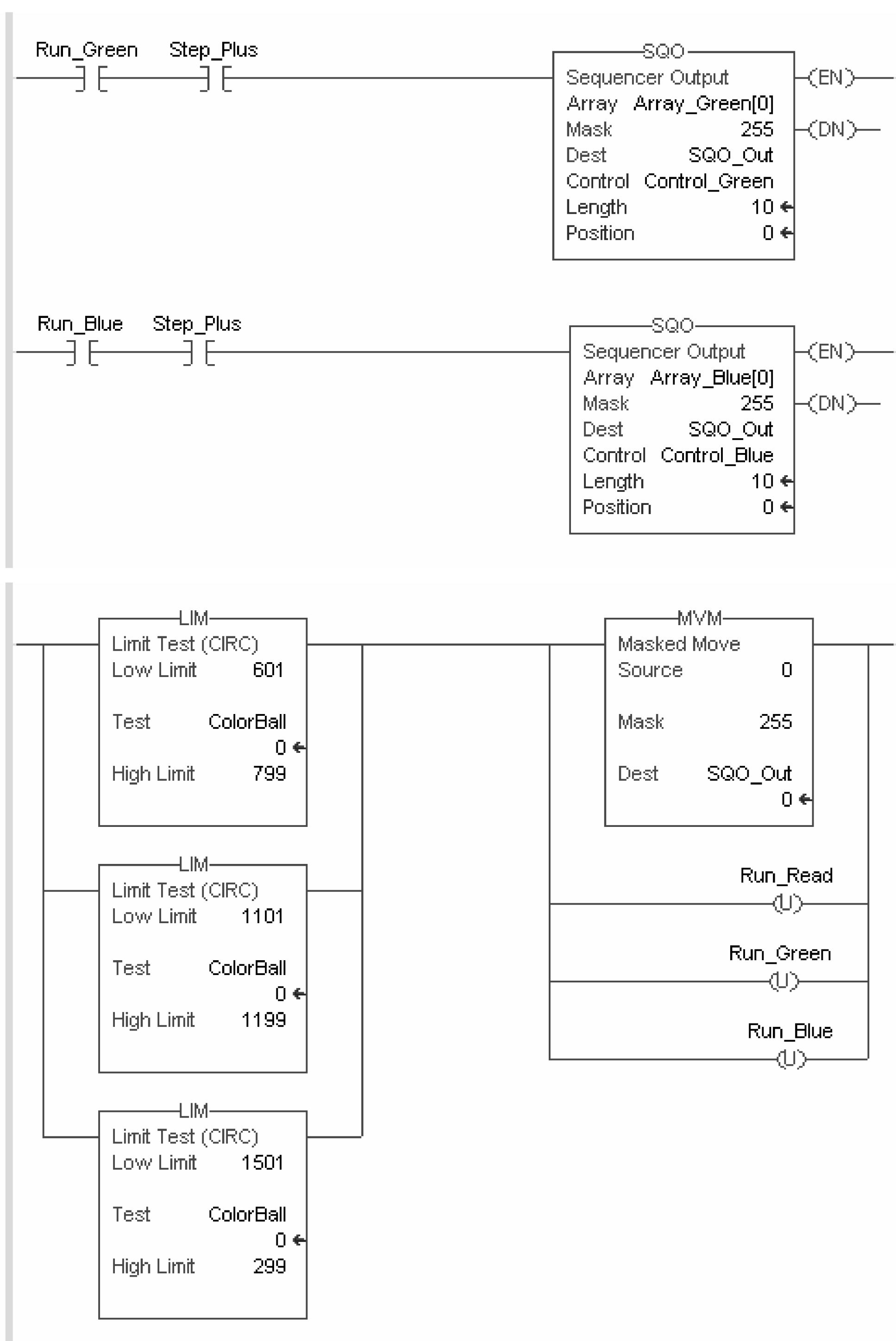


图 7-13　范围判断和执行的完整的梯级逻辑（续）

第 8 章

传送和转换指令的编程

梯形图的例程中，数据的传送和转换是比较活跃的一部分，控制系统信息处理的各个环节中，作为信息载体的数据常常需要变更类型、缓冲存放、范围变换或改头换面，针对不同的对象、不同的数据类型和不同的目的，要选择适合的传送和转换指令来完成，这些指令看似简单、理解容易，但还是有一些不容忽视的细节，在编程运用的时候需要认真对待。本章要讨论的就是如何正确地选用指令来完成不同的数据传送和转换工作。Logix 控制器的指令系统的传送和转换指令有：

- MOV　MVM　传送指令和屏蔽传送指令；
- COP　CPS　拷贝指令和同步拷贝指令；
- FLL　充填指令；
- BTD　位域传送指令；
- SWPB　字节交换指令；
- TOD　FRD　双整字转 BCD 码指令和 BCD 码转双整字指令；
- STOD　DTOS　字符串转双整字指令和双整字转字符串指令；
- STOR　RTOS　字符串转实数指令和实数转字符串指令。

8.1　传送指令 MOV 和屏蔽传送指令 MVM 的编程

我们首先讨论最常见的单一的字对字传输的 MOV 指令。它的操作对象限于字，可以是短整型字、整型字、双整型字和实数字，也可以是立即数。不管是源字还是目标字，特别提倡使用双整型字和实数字，因为它们是 32 位的数据单元，恰恰是 Logix 控制器数据处理的基本单元，是最为节省运行时间和内存空间的选择。同样的道理，在算术逻辑运算章节中已经有过解释，MOV 指令也是如此，在指令集的指令参数描述中，你仍然可以看到 MOV 指令的数据类型也是用黑体字表现了双整型字和实数字，这是推荐使用的数据类型。

MOV 指令是不能够直接对结构数据标签操作的，但可以对结构数据标签中的子元素操作，如果这个子元素正好是一个字的话。

相同的数据类型或不同的数据类型之间的数据传送，结果会是怎样的呢？现在我们将双整型字和实数字传送的 4 种组合编写 4 条指令，进行对比，如图 8-1 所示。控制器运行后观察的结果是：

- 实数字传送到实数字，不改变；
- 双整型字传送到双整型字，不改变；

- 双整型字传送到实数字，目标地址实数字，小数部分补零；
- 实数字传送到双整型字，目标地址双整型字，小数四舍五入到双整型字。

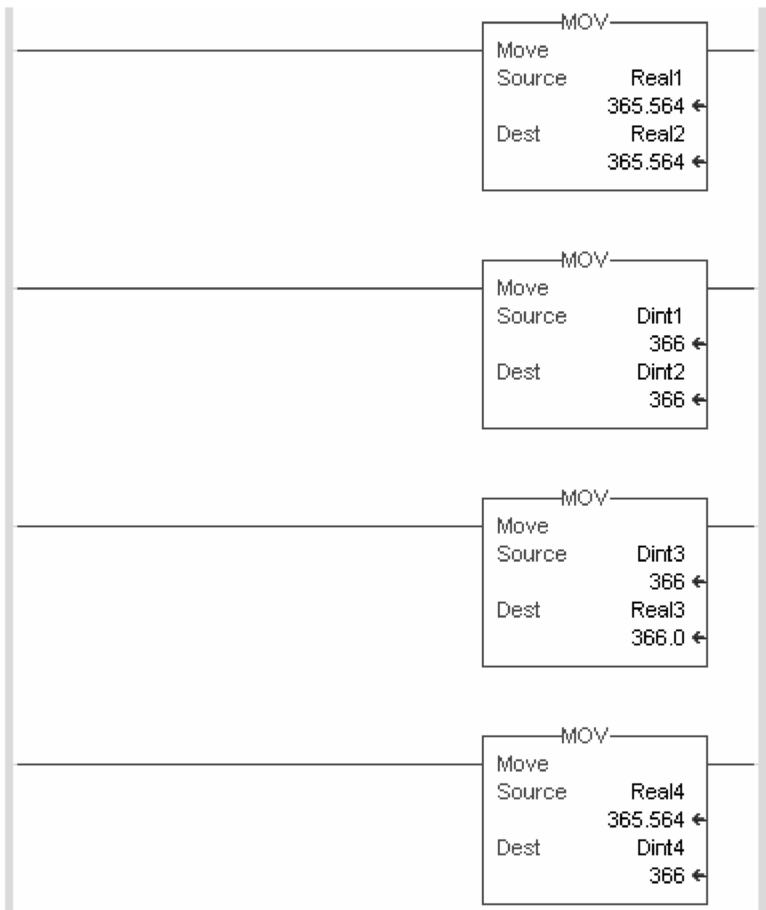


图 8-1　双整型字和实数字传送的 4 种组合

使用不同类型的数据传送，要注意小数点后面的数据处理，如果这个数正好是用于累积的话，有可能造成累积误差。积累误差的影响对于计算指令来说也会有类似的情况。

很多人编程时不加思索地给予 MOV 指令无条件梯级，这样可以令源标签和目标标签保持高度一致。如果你的需求源数据和目标数据无须时刻保持一致，建议你编写成梯级条件控制，只有必要时执行 MOV 指令传送数据，这样可以节约梯级逻辑扫描时间，减少无效或无意义操作，个别 MOV 操作虽耗时不多，但会积少成多，这是系统优化要注意的细节之一。有的时候不仅仅是节约了梯级扫描时间，甚至可以节约更多的系统资源，比如，在冗余控制系统中，减少数据刷新次数就可大大缩减冗余控制系统用于的冗余数据备份的时间，这可是非常宝贵的系统资源。如图 8-2 所示是受条件限制的 MOV 传送。

MOV 指令也可以直接将一个立即数传送到目标标签，但要注意到 MOV 指令所传送的立即数在默认的情况下是一个十进制的数。如果 MOV 传送的是一个状态的设置字，不妨借助于数据文件的 DINT 字来换算，先在二进制形式下将状态位设置好，然后转为十进制数，此

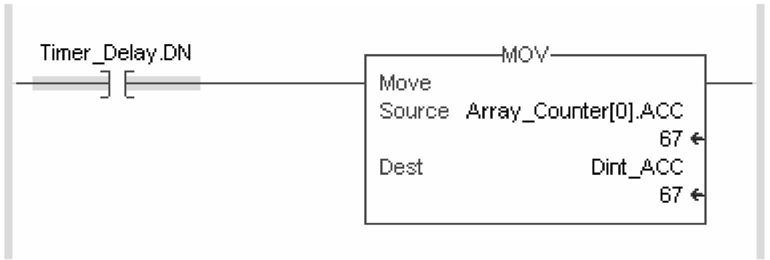


图 8-2　有条件的 MOV 传送

值即为 MOV 指令要传送的立即数。当然，如果特别地将立即数书写成十六进制和八进制的形式（16#26 或 8#46），指令也是接受的，但最好将数据库中目标标签的数据表达式也改为同样的数据形式，这样可以直观地看到传送的结果。通常标签默认的数据形式是十进制形式，数据形式的不一致可能会让你感到困惑。

关于数据表达形式的细节，平时人们不太留意，一旦面对这些问题却往往大惑不解，其实这就是一个计算机数据表达形式的规定而已，当你主观地用某种进制的数据键入时，数据单元的 32 位寄存器会客观地用 0 或 1 状态记录下来，当你换另外一个进制主观地去读，表达形式已经变了，但量值没有改变。如图 8-3 所示的例子是在十进制的形态下键入 45 后，分别在八进制、十六进制和二进制下的数据表达形态，读数变了，量值却没有变。

Value ←	Style
45	Decimal

Value ←	Style
8#00_000_000_055	Octal

Value ←	Style
16#0000_002d	Hex

Value ←	Style
2#0000_0000_0000_0000_0000_0000_0010_1101	Binary

图 8-3　不同进制的数据表达形式

MOV 指令的用法是比较灵活的，常见的还有传送一个 0 到目标标签，达到清零的效果。尽管清零已有专用的指令 CLR，它的执行时间更短，清零意义明确，也许是出于习惯，还是有不少人喜欢用 MOV 指令。我建议对一个字清零最好使用 CLR，这符合尽可能使用指令功能的原则，且表意明确，对比如图 8-4 所示。

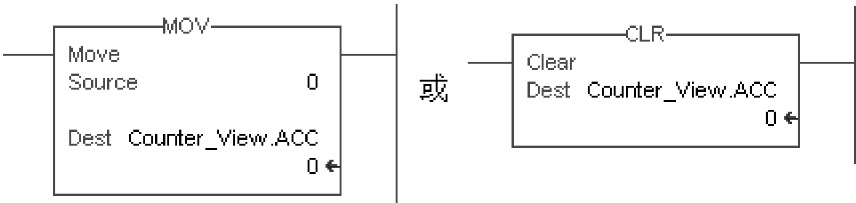


图 8-4　两条清零指令的对比

最常见的 MOV 指令的运用，还会出现在初始化的例程里，用于初始化字的传送，为某些标签数据设置初始值或者清零，用立即数传送到目标标签。系统中我们常常需要对某些特

定的数据块进行组态设置，如果直接在数据表中进行设置，万一出现扰动改变了数据就可能带来混乱，对于非项目开发人员来说，难以恢复原来的设置。有经验的项目开发人员为了提高系统的可靠性，会将这些要设置的数据用 MOV 立即数操作，直接将组态数据置入数据单元，通常在控制器的初始化例程中统一设置，这样，一旦出现设置数据混乱的局面，只需重新启动运行便可恢复正常。配合实例运用的数据设置我们将在后面的章节讨论。

屏蔽传送指令 MVM 主要用于局部地对字进行传送，只能对整型字传送，实数字不能传送。MVM 指令是用屏蔽码（可以是立即数或整型数标签）针对位来屏蔽的，一般来说，用于位传送的屏蔽。梯级逻辑编写如图 8-5 所示。梯级执行的结果是只将源标签的低 8 位传送到目标标签，目标标签的其他位不受影响，这两条指令分别使用了不同的屏蔽码的表达方式，二进制的屏蔽码较为直观，十六进制的屏蔽码较为简洁。对于按字节的屏蔽，用十六进制的屏蔽码表达会更简单。

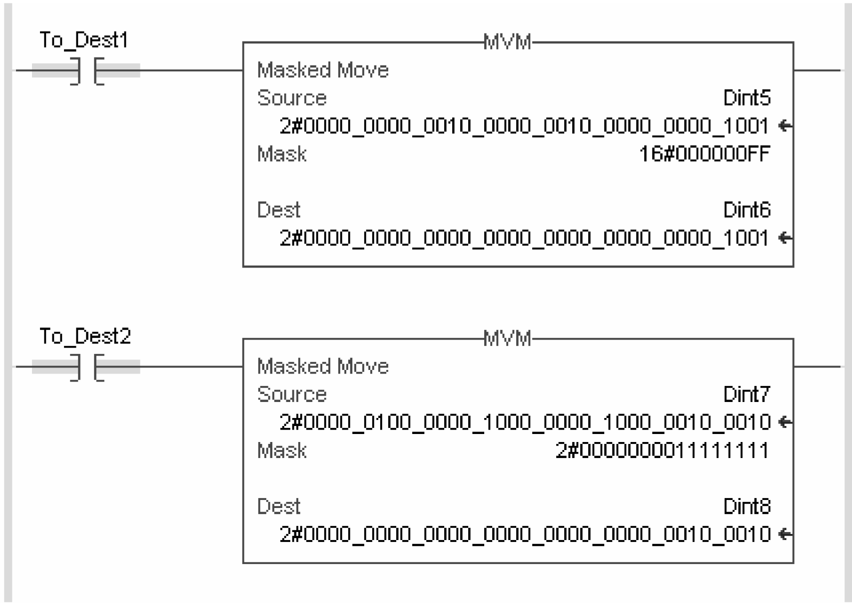


图 8-5　屏蔽传送的梯级逻辑

屏蔽码最终表现在屏蔽位上，屏蔽位决定数据是否能够传送，每个对应位的关系就像通过与门传递，对应位为 1，表示数据可传送通过；对应位为 0，数据被屏蔽不能传送，如图 8-6 所示。不仅仅是屏蔽传送指令 MVM 的屏蔽码才具有这样的含义，屏蔽码的屏蔽方式同样也适用于其他指令，以后遇到指令中的屏蔽码不再赘述。

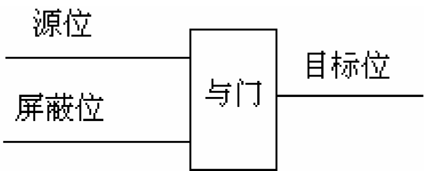


图 8-6　屏蔽关系图

MVM 指令经常会用于 I/O 模块的数据传送，因为这样的传送可以屏蔽无须传送的点，只对部分相关的点传送状态，避免了干扰和冲突。如图 8-7 所示就是只传送输入模块的低 6 位进入缓冲字 Buffer_SQI，防止了高位无关数据的干扰。十六进制屏蔽码的译读也很简便，屏蔽码 3F 一看便知是允许通过低 6 位。

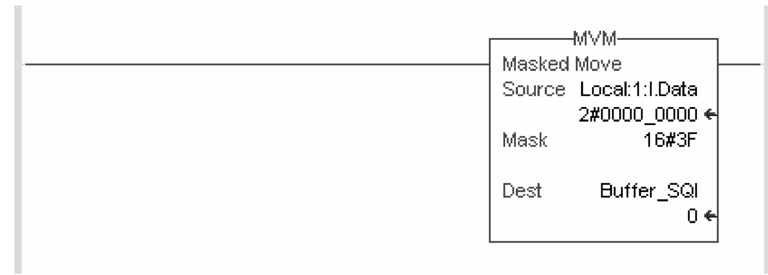


图 8-7　只传送输入模块的低 6 位进入缓冲字

8.2　复制指令 COP、同步复制指令 CPS 和充填指令 FLL 的编程

COP 指令是常见的一条指令，用来搬运各种数据十分方便。很少有人留意到，这是一条针对存储器操作的指令，它将源标签的数据按照存储器的 0 或 1 的状态，原封不动地复制到目标标签，起始标签元素和指定的长度确定了参加复制的存储器区域，它传送的数据必须是连续的，不可挑选的。如图 8-8 所示是常见的 COP 指令的梯级逻辑编写。这里源标签和目标标签都是相同数据类型的数组，这种情况下，对 COP 指令长度的设置是没有疑问的。

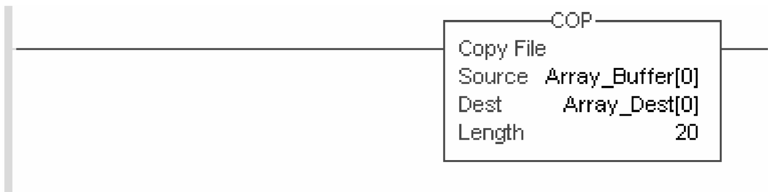


图 8-8　常见的 COP 指令的梯级逻辑

基于对存储器的操作，COP 指令操作既是适应性很强的数据传送，又是非常呆板的数据传送。适应性强指的是不同结构数据类型的数组标签都可以相互进行复制，呆板指的是只能将整块物理区域数据复制而无法挑选其中的部分数据。

由于可以在不同类型的数据之间进行复制操作，COP 指令参数的 Length 项便很有说法。指令参数的含义是这样描述的：COP 指令的长度指的是目标标签数组元素的个数，请注意是元素的个数，而不是存储器单元的数目，一个基本数据类型的标签或一个结构数据类型的标签都被称为一个元素。不同数据结构类型标签的数组耗用的存储器数据单元是不同的，而 COP 指令操作又是只认目标标签的数组元素个数的。请看一个不同数据类型数组标签之间的 COP 指令的运用实例，如图 8-9 所示。

这两条 COP 指令被编写在不同的地方，一条位于传送数据的例程，另一条位于接受数据的例程，它们的操作对象是相同的。前一条 COP 指令将 5 个计时器的结构数组标签复制到双整字数组标签，因为每个计时器的结构数据元素耗用 3 个双整字，所以必需 15 个双整字元素的存储单元才能存放，指令的长度要设为 15，以目标标签为准。后面一条 COP 指令是还原的，对调了源标签和目标标签，将 15 个双整字数组元素的标签还原到 5 个计时器数组元素的标签，目标标签只有 5 个计时器结构数据元素，长度设为 5 才是正确的。如果没有了解 COP 指令的长度的含义，想当然地键入长度 15，将造成错误，这种错误是常见的，这

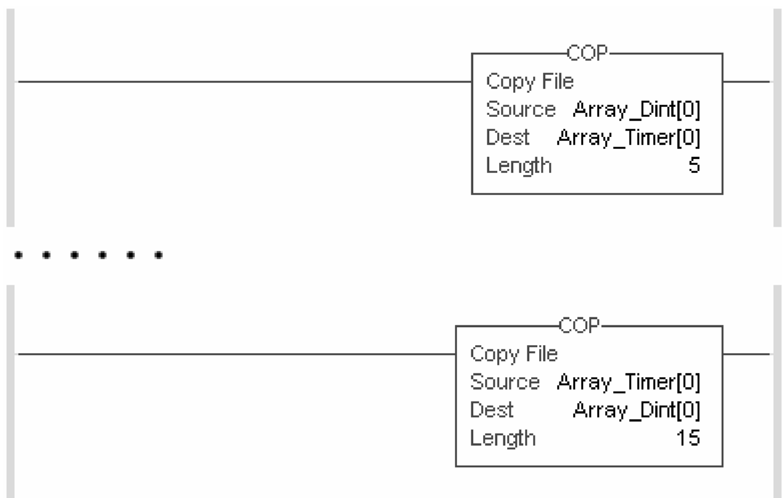


图 8-9　目标数据标签的数据类型决定了长度

请不要忽视指令参数的含义。

实际上，我们要传送的数据都是动态的数据，静态的数据是无须传送的，结构数据中总是同时包含了静态数据和动态数据，比如计时器结构数据标签的 3 个双整字，只有累加值 ACC 是变化的数据，是值得传送的，其他两个字预置值和状态量是不变或自变的设置数据，传送是没有意义的，无端地消耗了资源。只需要传送计时器结构数据标签的 ACC 值，如果使用 COP 指令来完成，显然是一种浪费，有 2/3 是无意义的数据传送。节省的做法是从结构数据中检索需要传送的子元素，集合在一起传送，这样的执行动作，COP 指令是难以胜任的，最合适的是采用 FAL 指令来完成。关于 FAL 指令的运用将在下一个章节详细谈到，恰恰是这条指令，弥补了 COP 指令的不足。

一般来说，用 COP 指令传送结构类型数据标签，往往为了整片数据的复制，请留意长度的设置。

早期，关于 COP 指令的运用也有非常有趣的技巧性的做法。例如，数据采集堆栈存放的处理，在一个数据堆栈的数据区域中，每进入一个新的数据，之前已存放的旧数据块整体往前移动一个，不断地推陈出新，该怎么做呢，请看如图 8-10 所示的梯级逻辑。

这条 COP 指令将数组 Array_Move_Buffer 的元素 1 为首址的数据块，复制到同一个数组 Array_Move_Buffer 的元素 0 为首址的数据块，操作长度为 20，这样的复制是含有覆盖的作用的。执行完毕的结果如图 8-11 所示。可以看到，相当于全体往前移动了一个数据单元，配合先执行的 MOV 指令，在元素 20 的字（第 21 个字）中事先装入新的数据，综合执行的结果是：装入一个新数据，并移动全部数据，推掉最前面一个数据，从而保持了最新采集的 20 个数据，注意被操作数组创建的长度要超过 21 个。

注意这里给定的梯级条件，只要 Move 位状态的梯级条件成立，COP 指令每次扫描都会执行，所以要确保每逢 Move 位状态跳变一次才能执行一次 COP 指令，必须在梯级条件后面加上 ONS 指令给予限制。

如前所说，当梯级条件 Move 位状态跳变，MOV 指令装载一个新数据，COP 指令使得整体数据前移一个数据单元，如果使用一个具有定时功能的自复位计时器的完成位作为梯级条

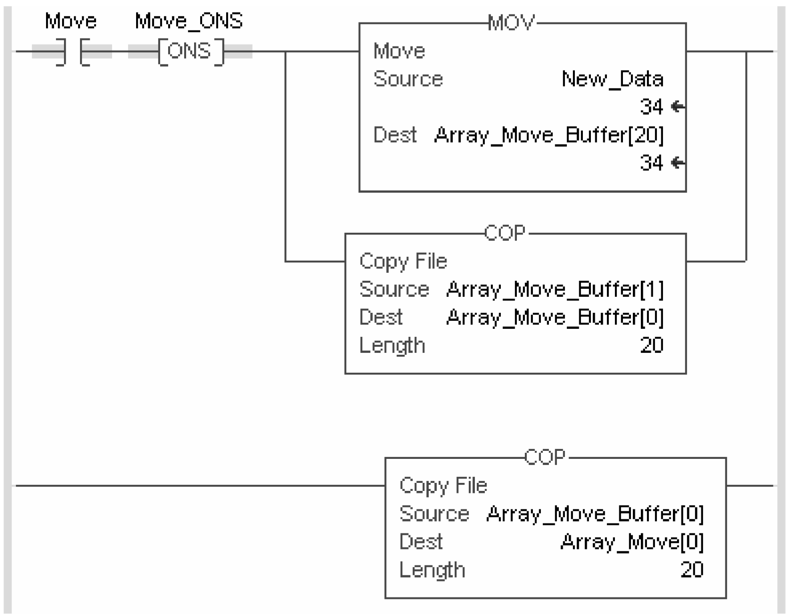


图 8-10　堆栈式复制的梯级逻辑

件，编写梯级逻辑如图 8-12 所示，数组的数据将自动地周期性移动，New_Data 数据单元中的数据则提供填入的新数据。这里，由于我们了解自复位计时器完成位 DN 只会存在一个扫描周期，所以 COP 指令能确保只执行一次，此处不用 ONS 指令限制也是可以的。最后，将存放在缓存区的移动传送成功的数组，一次性复制到最终标签 Array_Move 数组。

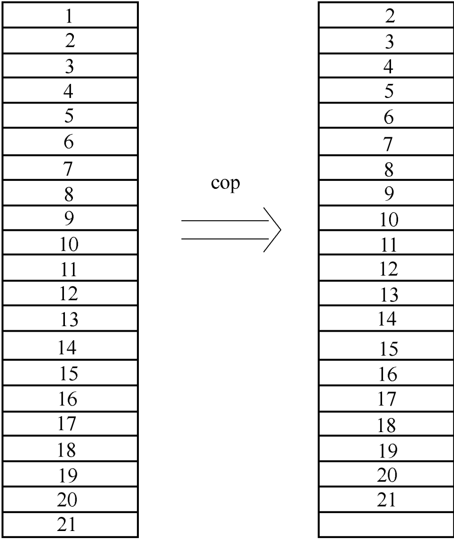


图 8-11　COP 指令执行完毕的结果

这种做法很像后面要讨论的堆栈数组的处理，也许堆栈指令 FFU 的宏内部操作就含有 COP 指令这样的执行过程。早期的可编程产品没有堆栈数组处理的指令，要满足这样的需求，就是采用这个办法，这也算得上编程技巧的处理了。

CPS 指令的用法和 COP 指令是一样的，惟一不同的是执行过程，这是一条优先级别很高的指令，控制器中的其他任务执行触发都不能中断这条指令的执行。一般的 COP 指令在执行的时候，都是一个字一个字的传送，有可能在传送的操作过程中被优先权更高的执行动作中断，等恢复操作接着传送数据时，也许源标签的数据已经被刷新了，后面一段传送的则是刷新后的数据。这就出现了 COP 指令执行完后，数据块的数据是新旧混杂的情况。

特别是当外部数据传送到内部数据，数据长度大过 32 位，也就是多于一个基本数据单元时，由于背板 CPU 的优先权高过逻辑 CPU，背板 CPU 完全可能在 COP 指令执行过程中请

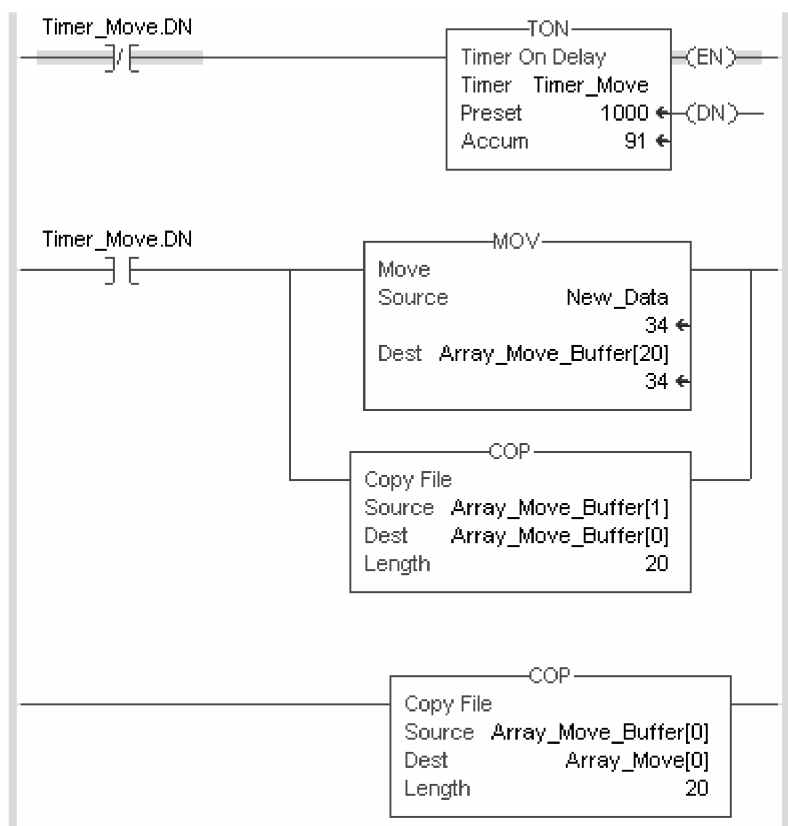


图 8-12　数组周期性移动的梯级逻辑

求数据的刷新，因而新旧数据混杂的可能性非常大，强烈建议使用 CPS 指令，这样在数据复制过程中不会被背板 CPU 数据刷新的请求中断。如图 8-13 所示是 Consumed 数据的传送，使用 CPS 指令有效地防止了在同一个数据块中新旧数据的混杂。

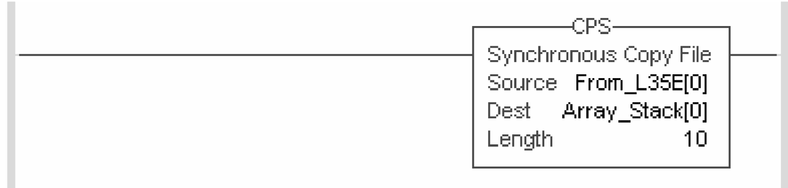


图 8-13　Consumed 数据的传送使用 CPS 指令

总之，只要数据复制是在不宜中断操作之处，有新旧数据混杂风险的情况都建议使用 CPS 指令。

前面，我们曾经提到用 CLR 指令清除一个字，如果需要清除一个数组，则可以采用充填指令 FLL 来完成。FLL 指令的执行可将一个数据单元或一个立即数充填到整个数组，通常用来实现数据的初始化。如图 8-14 所示编写的梯级逻辑是对一个数组的清零，将一个为 0 的立即数充填到数组目标标签，一般情况下，这个数组可能是一个操作过程的中间状态量的集合，在程序运行之初，要将上次运行的所有的残留状态清除。有现场调试经验的工程师都非常清楚前次残留状态对系统当前运行的影响，而不敢掉以轻心，这些现象是不定的，有时

测试几次都不会暴露，为了不留下隐患，所以宁愿多费些功夫要清除，以免系统出现不可预料的结果。

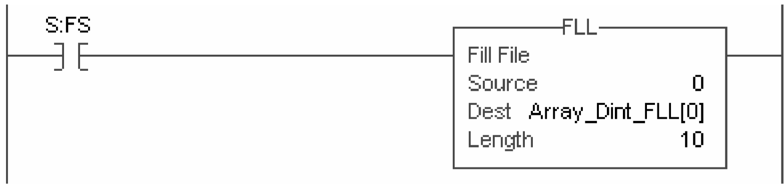


图 8-14　对一个数组的清零

在有的实用例程中你可能看到，FLL 指令所做的不是对一个数组清零，而是对所有的字赋予一个 1，编写梯级逻辑如图 8-15 所示。这是一种初始化数据的方式，有时是为 SFC 的步操作设置初始步，前一次运行停止，SFC 或许停止在中间的某一步，这也是残留状态，这个初始化的操作令每个 SFC 流程都回到初始步。有关初始步的问题我们在 SFC 编程中再详细讨论。

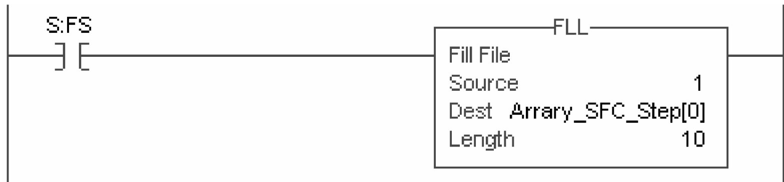


图 8-15　初始化数据

总而言之，充填指令 FLL 的使用较多情况是完成初始化的操作，我们常常可以在初始化例程中看到这条指令。

FLL 指令跟 COP 指令有相似之处，虽然操作对象是数组，也要指明长度，但是没有指定一个专用于数组操作的控制结构数据标签，也没有指令执行过程的各个状态位。它们的操作是对数组的连续操作，没有停顿和指针位置，它们更像是数组的数据传送。关于控制结构数据标签，后面章节的内容马上就要用到了。

8.3　位域传送 BTD 指令和字节交换指令 SWPB 的编程

以上的传送都是数据单元的传送，不管是字对字的传送，数组对数组的传送，抑或结构数据对结构数据的传送，在传送的时候，每个元素都是毫无选择地位对位地进行传递，如果字与字之间需要错位传送就不得不使用位域传送指令 BTD 了。在某些情况下，这可能是最好的解决方案。

BTD 指令是选取一段位域传送到一段指定的位域，位域的传送可以是字内的传送，也可以是字与字之间的传送，传送源和传送目标不得跨越字的范围。

当 BTD 指令的执行将一段位域的位置从一个字搬到同一个字或另一个字的位置，指令参数的源字和目标字为同一标签名将完成字内传送；源字和目标字为不同标签名将完成字间传送。被操作位域的长度要适合目标字的放置空间，如果目标字空间不够，将造成前端位域数据的丢失，溢出的位是不会转移到下一个字的。

如图 8-16 所示的指令参数键入所表示的是，在梯级条件成立时，指令执行将源字 Dint-

Source_BTD 从 03 位起始的 10 个位传送到目标字 DintDest _BTD 从 05 位起始的范围，这是一个字间传送的例子，如图 8-17 所示。

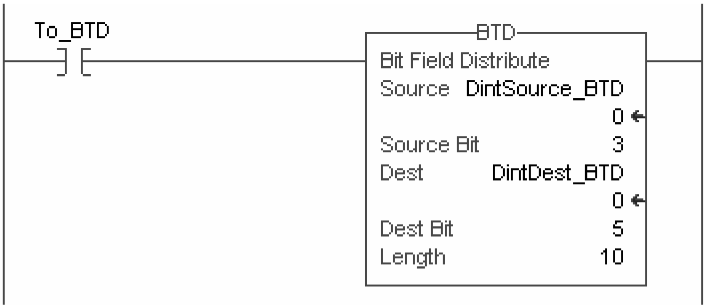


图 8-16　字间的位域传送

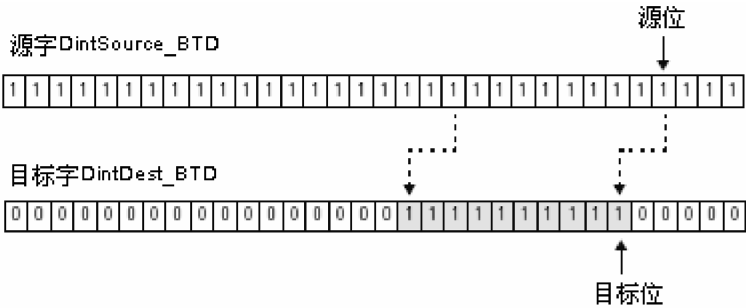


图 8-17　字间传送示意图

这里提到的字实际上指的是基本数据单元，不同时期的控制器产品，数据处理的基本单元是不同的，这取决于 CPU 的处理芯片，最早期的 PLC，处理芯片是 8 位，后来发展到 16 位，现在我们使用的 Logix 控制器 CPU 芯片处理的是 32 位。32 位理所应当地成为基本数据处理单元，这些都是我们要关注的，这就是为什么我们前面编程一直强调要使用 DINT 和 REAL 数据类型，只有直截了当的使用基本数据单元，才是最节省计算机资源的，不论是时间还是空间。

但是对于通信而言，基本传递单位却是字节，所以在建立通信数据时需识别字节的放置关系。说起来很有趣，欧洲产品的习惯在字节传送到达时，按照从低字节到高字节的顺序排列来放置；北美产品的习惯在字节传送到达时，按照从高字节到低字节的顺序排列来放置。不幸的是，这种放置顺序在产品技术手册中并不常出现或没有特别强调，当这两种不同习惯的产品相遇，现场调试的工程师们便大受其害，连呼上当。所以无论使用哪家公司的产品，一定要留意这点，当数据传送出现谬误时，首先应当想到可能是字节的顺序出了问题。如果凑巧找不到资料，最好的办法就是测试一下。

不管是使用 Logix 控制器作为信息发送者要与第三方设备通信需要满足排列顺序的规则，或是 Logix 控制器从第三方设备接收到的字节顺序不符合 Logix 控制器数据的排列顺序，字节排列顺序的重新安排都必不可少，编写梯级逻辑来改变字节或字的排列顺序的工作就责无旁贷了。

假定我们要不断地传送一个双整字 DINT 的数组的处理结果到一台第三方外部设备，并

要求将每个双整字的字节顺序颠倒了传送出去。

如图 8-18 所示，将源字的最低字节传送到目标字的最高字节；源字的次低字节传送到目标字的次高字节；源字的次高字节传送到目标字的次低字节；源字的最高字节传送到目标字的最低字节。注意到，这里所说的一个双整字右边是低字节，左边是高字节。

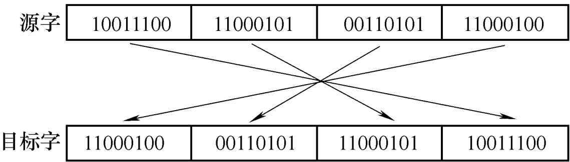


图　8-18

假定我们需要处理的数据块有 40 个双整字，我们将编写梯级逻辑为 40 个双整字转换字节的顺序，采用循环 40 次执行的办法来完成。这可能是比较麻烦的处理方式，编写的梯级逻辑如图 8-19 所示。

编写 4 条 BTD 指令来分别完成 4 个字节的顺序改变，即一个双整字的处理，用来实现图 8-18 中的传送。请注意指令参数源字和目标字的每个起始位都在字节的起始位，这是字节的位置调整。考虑到这是跳转的循环操作，BTD 指令前面无须设置梯级条件。BTD 指令的处理对象是数组元素号所指向的字，数组元素号采用间接寻址的方式，利用指针加 1 的修改来实现，随着指针的增加元素号偏移，从 0 开始到 39 结束，40 个元素的数组每个元素字逐一被处理。

一个双整字处理完毕，对循环的指针 Position_BTD 加 1，进入下一个循环，处理下一个元素字。每次进入循环之前，要经过比较判定决定是否跳回到前面继续循环，直到指针为 40 时，表示该数据块处理完毕，不再跳回前面的梯级。这里要注意，判断条件是小于 40，也就是 0～39 都会跳回前级处理，待第 39 字处理完毕，指针加 1 为 40，不再满足循环的条件，离开跳转循环。

离开跳转循环后，程序扫描下一个梯级，进行最后的数据块传送处理。此时等于指令 EQU（也可使用大于等于指令 GEQ）的 Position_BTD = 40 的判断能够满足梯级条件，将处理缓冲区数组的数据 COP 给目标数组，然后清零指针 Position_BTD，准备下一轮的处理。如果没有清除指针，非但不能进行下一轮的处理，控制器还将报错，并呈现故障状态，停止运行。

显然循环指针同时也是间接寻址的元素号，每个 40 个字长的双整字数据块的处理都是从 0 元素开始的，不但在每个循环结束的时候需要对指针清零，在程序的初始化例程中，也要编写清除指针的操作，如图 8-20 所示。因为我们不能确定上一次程序停止运行时，指针是否在 0 的位置，当第一个 BTD 指令的梯级开始被扫描时，双整字数据块必须从 0 元素开始处理。

这里我们讨论一下，为什么顺序改换的中间处理结果要暂存在缓冲区 Array_To_Buffer 数组，待所有的处理全部完成之后再复制给目标数组。因为在执行数据顺序交换处理的梯形逻辑的时候，如果逻辑扫描被中断，正在处理的数据必然是残缺的，也许影响使用，在使用条件苛刻复杂的地方，有时甚至连 COP 指令的执行都有可能被中断，这时就强调要使用同步复制指令 CPS 了。

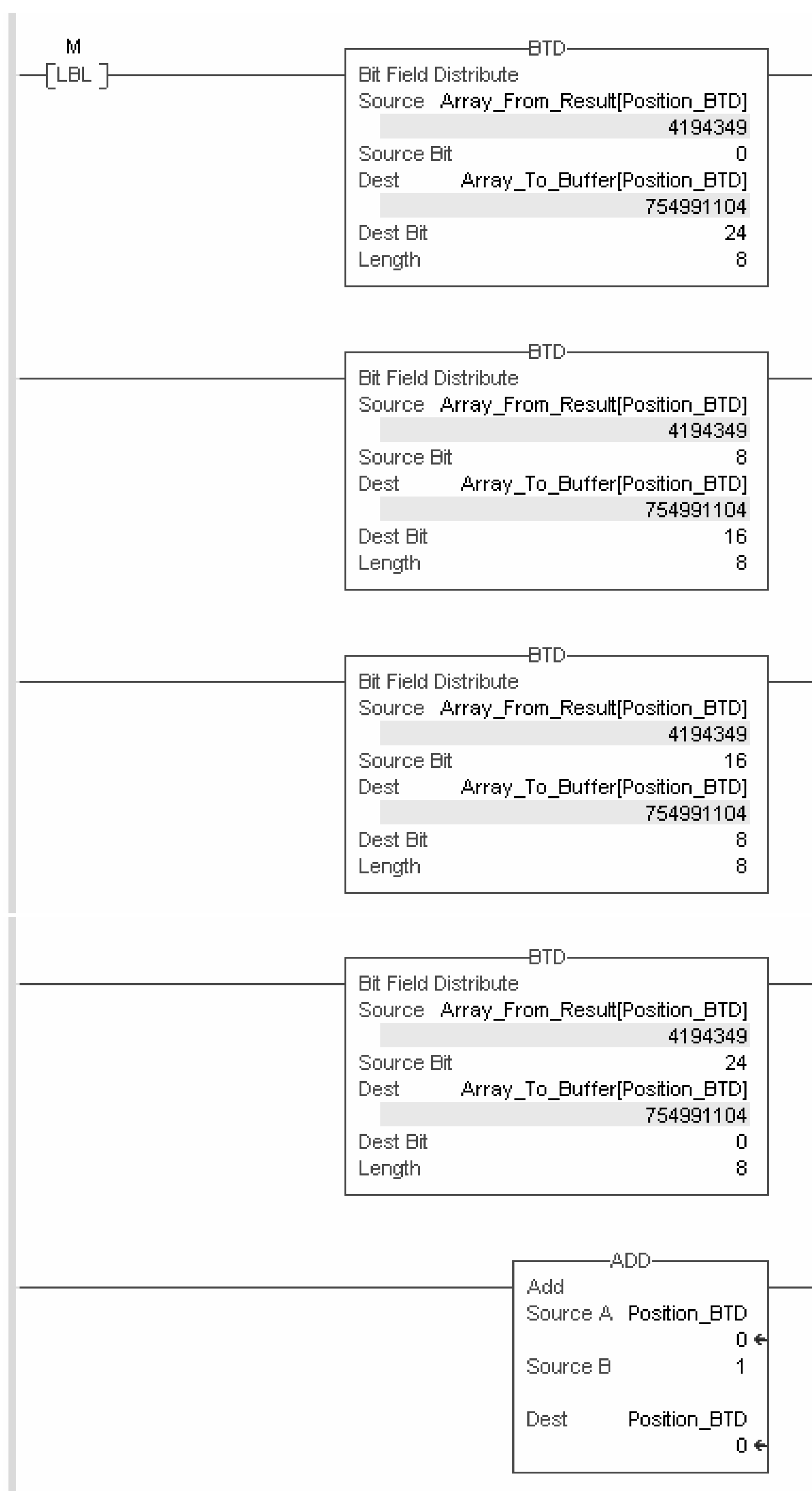


图 8-19　改变字节顺序的循环操作

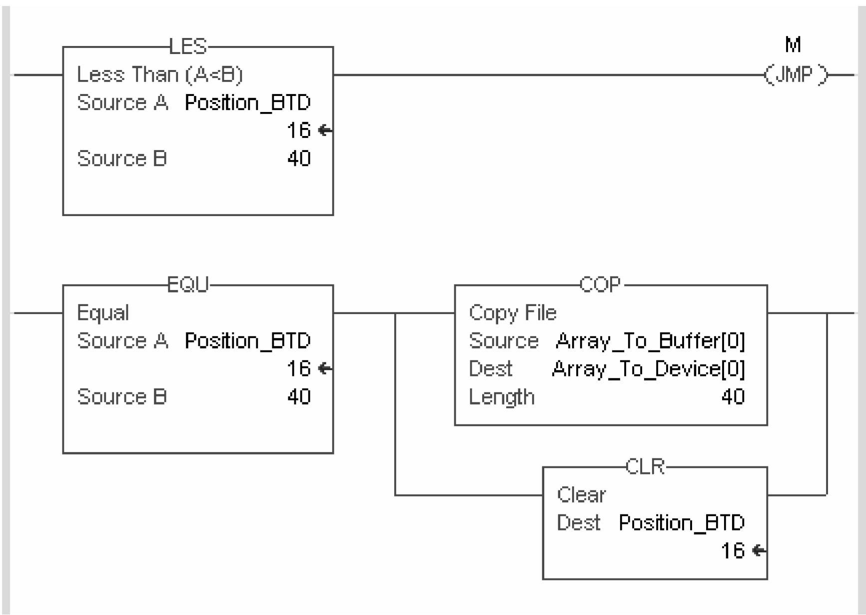


图 8-19 改变字节顺序的循环操作（续）



图 8-20 清除指针的操作

将正在处理的一批数据，尤其是关系紧密的数据，暂时存放在缓冲区，等待处理结束后再复制给目标数据，之后的任何执行代码的数据引用只使用目标数据，这也是编程的良好习惯之一，可以帮助你避过数据新旧混杂的陷阱，否则你将看到程序运行结果是不可预料的。

如图 8-19 所示的梯级逻辑执行结果，改变了字节的顺序，如图 8-21 所示，摘取了被操作数组的前 3 个双整字，对比 BTD 指令执行前后的数据存在状态。

以上是早期的字节换序的解决方案，第二代 PLC 产品不得不利用 BTD 指令来解决字节换序的问题，需要编出不止一条梯级来完成，Logix 控制器则增加了一条指令专用来做此事，那就是 SWPB 指令，即字节交换指令，该指令有一个参数为 Order Mode 的选项，可确定三种交换形式：

- 选项：REVERSE　颠倒顺序，如 ABCD 转为 DCBA；
- 选项：WORD　字交换，即单整字的交换，如 ABCD 转为 CDAB；
- 选项：HIGH/LOW　单整字的高低字节交换，如 ABCD 转为 BADC。

一般来说，现场设备使用单整字的情况比较多，所以字节顺序更改还有针对单整字的，这三种交换形式，足以处理字节换序的各种情况。

编程时指令参数键入的三种情形如图 8-22 所示。这里为了表示清楚，特地将 DINT_Byte、Result_Reverse、Result_Word、Result_HighLow 的数据标签选用 ASCII 码数据表达形式。

BTD指令执行之前

Name	Value
+ Array_From_Result[0]	2#0000_0000_0100_0000_0000_0000_0010_1101
+ Array_From_Result[1]	2#0000_0000_0000_0110_0000_0000_0011_1000
+ Array_From_Result[2]	2#0000_0100_0100_0000_0010_0001_0101_1001

BTD指令执行之后

Name	Value
+ Array_To_Buffer[0]	2#0010_1101_0000_0000_0100_0000_0000_0000
+ Array_To_Buffer[1]	2#0011_1000_0000_0000_0000_0110_0000_0000
+ Array_To_Buffer[2]	2#0101_1001_0010_0001_0100_0000_0000_0100

图 8-21　BTD 指令执行前后的数据对比

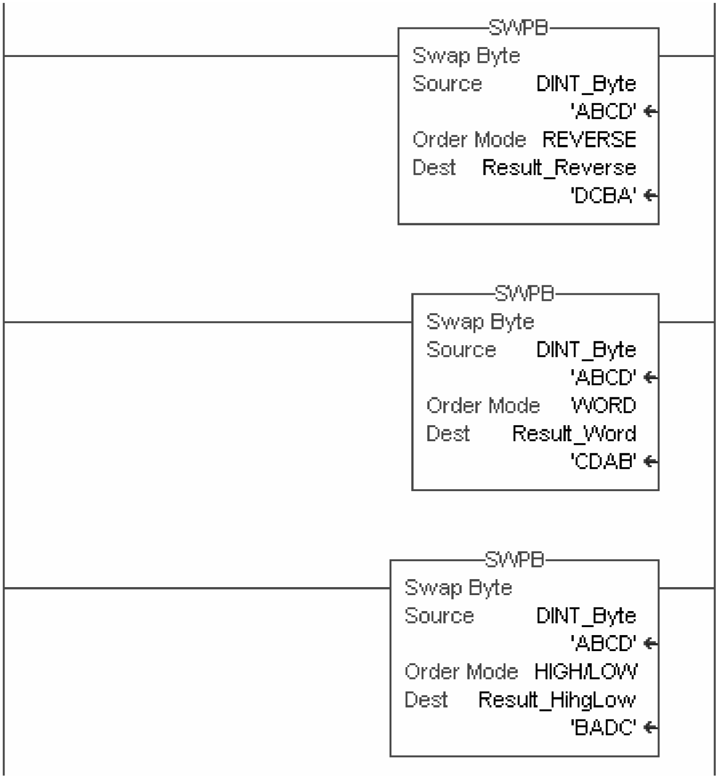


图 8-22　字节换序的三种情形

刚才的字节换序的需求实例就可以通过简单地编写 SWPB 指令解决了，梯级逻辑编写如图 8-23 所示。

仅 1 条 SWPB 指令就代替了需要 4 条 BTD 指令才能完成的工作，并且清楚明了了字节顺序的安排。这里，我们又一次见证了实际需求转为系统指令功能，无须编程技巧而使用指令功能直接实现。这并不能说明 BTD 指令就没有意义了，就字节顺序调整而言，SWPB 指令可以代替 BTD 指令，BTD 指令仍可用来完成不按字节调整的位域变化。

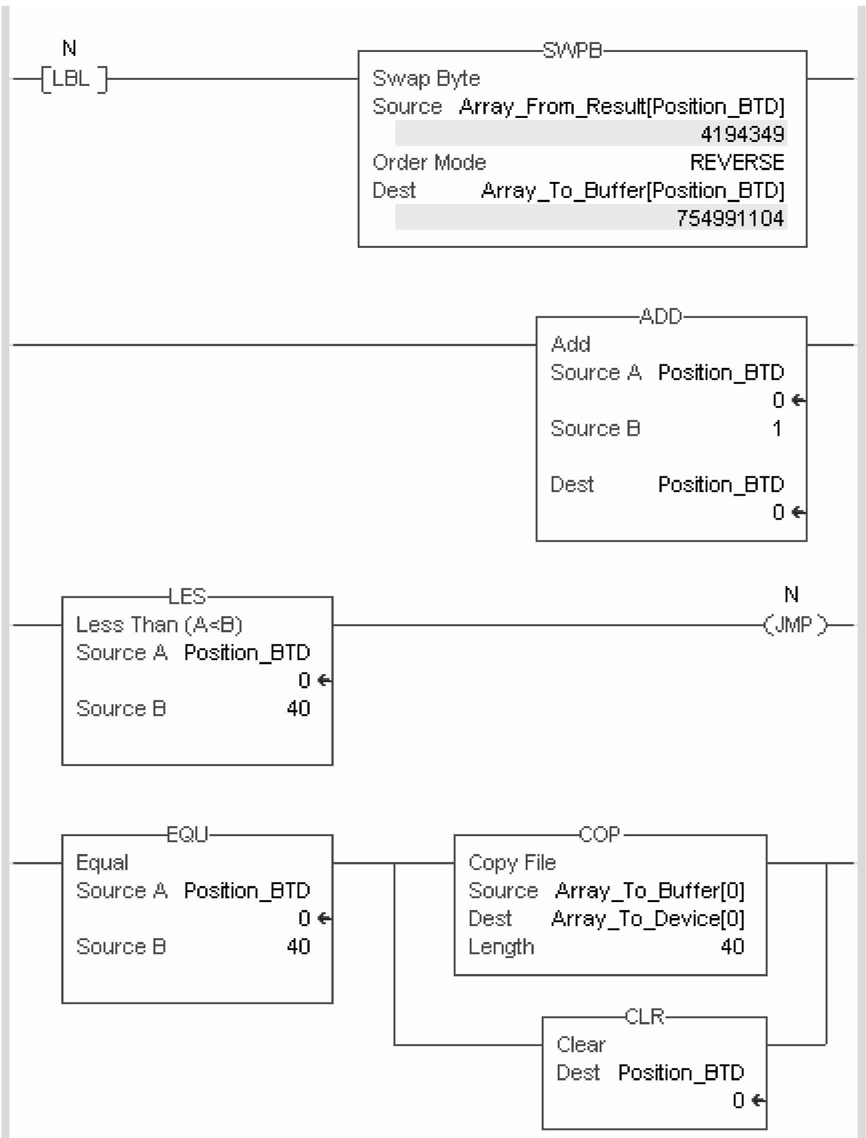


图 8-23　使用字节交换指令的梯级逻辑

8.4　转换指令 TOD/FRD 和 DEG/RAD 的编程

尽管当今的人机界面都是智能的人机界面，如 PanelView Plus，所有的数据都是通过网络通信，并直接设置数据，可以选择任何基本数据类型作为操作数。但是涉及 TOD/FRD 指令的讨论，不得不了解早期的人机界面。

可用来传递数字的，跟 PLC 结合的最早的人机界面，恐怕要算 BCD 码（即 8421 码）的数字组件了。这些数字组件是硬件，直接跟离散量模块连接，通过信号的输入/输出转换为数字代码，这是数字组件信号级别的人机界面。

早期的信号人机界面通常是离散量输出模块用 4 位输出组合对应数字组件的 7 段数码显

示来表达一个数字；离散量输入模块用数字组件拨盘位置对应 4 个输入点的组合来表达一个数字。16 点的 I/O 模块可以表示 4 位数的 BCD 码，其数据范围可表达 0 ~ 9999，通过对离散量模块的 I/O 扫描获取 BCD 码或传送 BCD 码，放置在处理器数据表中的 BCD 码则参与了数据的处理。

了解早期工控产品的工程师都知道，内存数据的表达形式分为几种，如图 8-24 所示。数制的数据处理是当今开发人员较为熟悉的形式，码制的处理由于智能人机界面的快速发展而变得较为生疏了。

数制和码制最大的区别是数字可以运算，码制不可以运算。来自 BCD 的码制是用来表达十进制的数字，很容易被人疏忽当做数字，不加思索地拿去运算，系统是不会提示这个错误的，并且会完成这个运算，其隐含的陷阱是把这个数字当做十六进制来处理，进位和借位遵循其规则，结果自然是失之千里。不要去责怪控制系统不能识别数制或码制，这原本就是人们的主观认识，机器只会客观地识别 0 或 1 状态，并把 0 或 1 状态换作它要处理的数据类型。

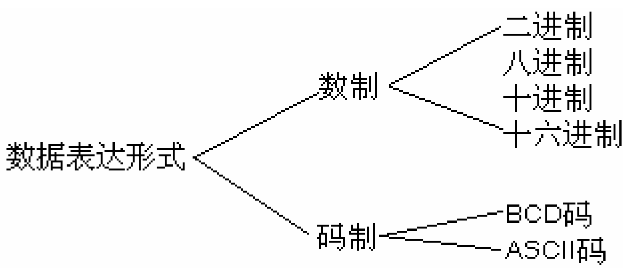


图 8-24　内存数据的表达形式

切记！所有的码制都要转换为数制才能进行运算。

TOD 和 FRD 就是解决换算问题的转换指令，TOD 是将十进制数转换为 BCD 码；FRD 是将 BCD 码转为十进制数。如图 8-25 所示编写的两个梯级把来自拨盘开关的 BCD 数据转换为待处理的十进制数据；把已处理好的十进制数据转换后送到外接的 LED 部件上显示数字。

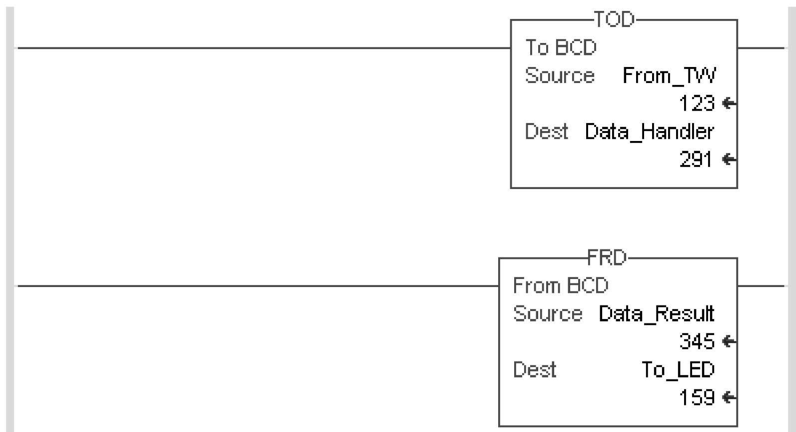


图 8-25　BCD 码转换的梯级逻辑

指令非常简单，但是一些细节不可忽视，我们已经习惯在指令的标签底下（蓝色箭头所指）读到相应的数据，这两条指令的 From_TW 和 To_LED 标签下读到的数据是没有意义的，因为系统总是按照二进制的权位换算成十进制数来表达，此处仅仅是 BCD 码的寄存器状态，按二进制权位换算出来的用十进制数的形式所显示的数字与所要表达的 BCD 码的数字毫不相干。在 Logix 控制器的数据表中，已经没有 BCD 码的数据形式，所以不可能在数据

表中改为相应的数据形式，并在指令参数上获得相应的显示。

这种情形不止一次地出现在其他指令中，千万不要把指令标签地址中所表达的数据想当然地看成跟指令有关联的数据，那不过是计算机机械呆板的一种表现而已，虽然很多情况下是正确的，但还是要酌情接受，不能一概而论。当你发现数据不是你想要的那个，不要武断地认为程序编写有误，盲目地去查程序而白费时间，而是要仔细考虑，系统将怎样来表达，或者去细读指令参数的含义或规定。

现在新配置的系统中这样的数字部件几乎消失，所以 Logix 控制器在数据库中取消了 BCD 码的表达形式，但保留了这两条转换指令。如果升级改造的控制系统仍有这样的数据采集进来，我们是无法在数据表中直接读到的。我建议你将这个数据改为二进制的表达形式，然后每 4 位分段地译读出来。同样的道理，如果我们需要将数据以 BCD 码的形式送出，最好改为二进制表达后再来分段进行设置。

TOD 指令还有一个用处就是可以用来设置初始值，如果这个初始值恰恰是要求用 BCD 码设置的话，正好我们在数据表里面直接设置甚为不便，当然也有考虑可靠性的因素。PLC5 模拟量模块 1771-IFE 的组态数据块，关于通道的定标数据要求用 BCD 码设置，在为组态数据块设置的 37 个字中，关于通道定标值的设置部分可以编写梯级逻辑如图 8-26 所示，这里只截取了通道 0 和通道 1 的设置。这跟使用 MOV 指令预设值是一样的做法，不过设置的是 BCD 码罢了。

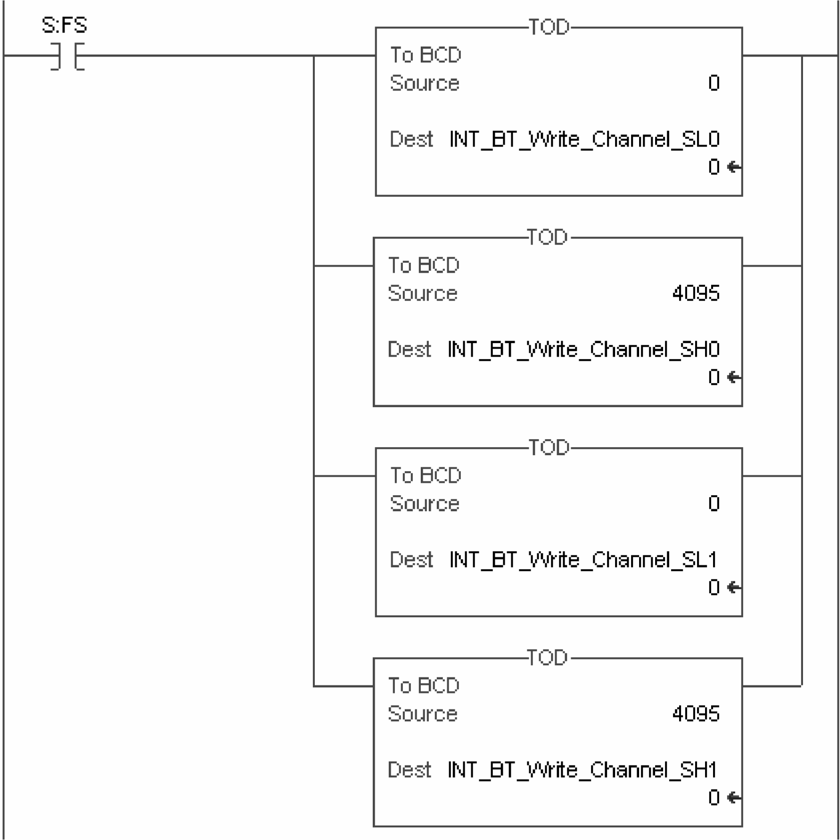


图 8-26　通道定标值 BCD 码的设定

TOD 指令和 FRD 指令跟 MOV 指令相似，也是立即数的传送，不过是将立即数从十进制转换为 BCD 码并传送到目标地址。

可以观察一下这几个梯级执行后的显示结果，如图 8-27 所示，指令目标参数表面所读到的数据 16533 对于我们想要的 BCD 码的 4095 是匪夷所思的，但是不必担忧，BCD 的数据已经被准确地送入了数据单元中。

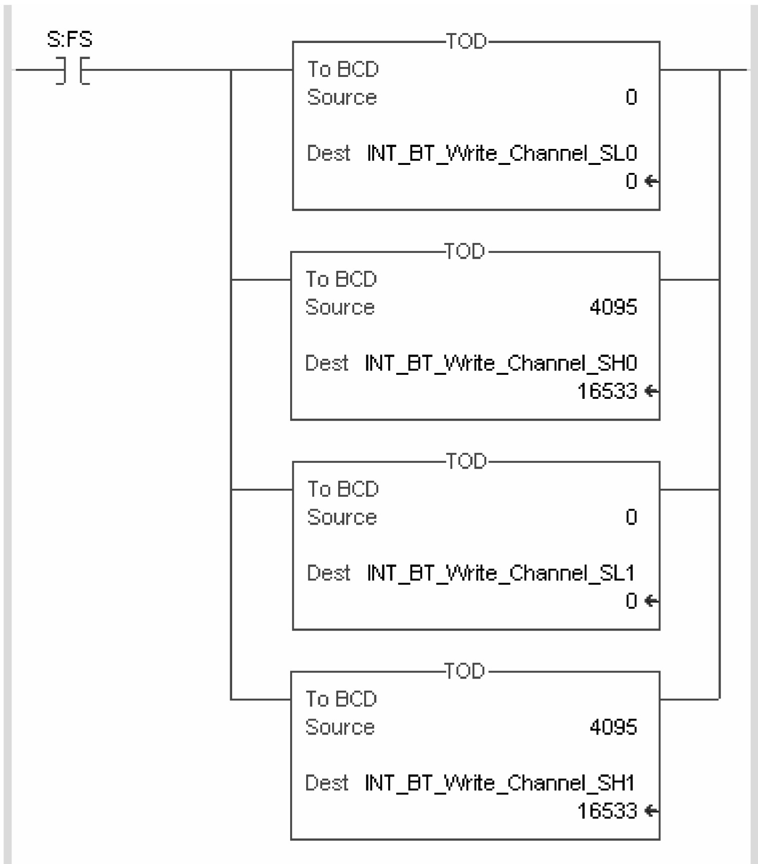


图 8-27　BCD 码设定后的显示

TOD 指令和 FRD 指令单一的转化操作也可以在表达式中直接引用，比如说对 CPT 指令和 FAL 指令表达式的运算结果进行转换，后续实例中我们将看到这种应用。

Logix 控制器在数据库中取消了 BCD 的数据表达形式，告别了渐渐远去的人机界面硬件数字组件方式。我们庆幸免去了难堪的 BCD 码的纠缠，却又陷入了 ASCII 码的处理。显然，Logix 控制器更为重视 ASCII 码的数字信息处理，在一组 ASCII 码信息处理指令中增加了几条有关数字和 ASCII 码相互转换的指令。

弧度转角度指令 DEG 和角度转弧度指令 RAD 也是一组转换指令，对编码器数据采集的处理有时会用到弧度和角度之间的转换。这里我们看一个实例，将数据转为弧度数值，作为提供给正弦指令 SIN 的源数据，梯级逻辑编写如图 8-28 所示，这是我们在三角函数的计算中曾经用过的。

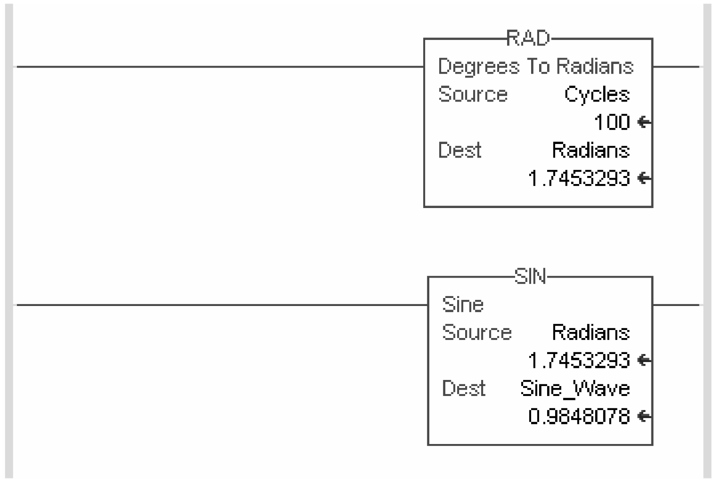


图 8-28　为正弦计算提供弧度值

8.5　ASCII 码转换指令 STOD/DTOS 和 STOR/RTOS 的编程

ASCII 码的信息载体比起 BCD 码组件是更进了一步，至少脱离了信号传送的模式。ASCII 码是来自于通信设备的数据串，通常是第三方 ASCII 设备通过标准通信口串口跟控制器来交换数字，常见的设备如条形码阅读器。

经过串口读写处理的 ASCII 码字符串的数字放置在数据库中，也许因为 ASCII 码每个数字占用了一个字节，不能直接译读，也许从串口获取的过程足够复杂印象深刻，人们对 ASCII 采集来的字符数据似乎不大容易被混淆为数字，而是很认真地想转换为数字。过去，这曾是很麻烦的一件事，需要编制一段较为复杂的梯级逻辑去解决转换问题。

较晚版本的 Logix 控制器关于 ASCII 码的处理增加了一些指令，其中就包括了 STOD 和 DTOS 这两条指令。STOD 是将字符串转为双整字，DTOS 则将双整字转为字符串。现在，将数字的 ASCII 码字符串转为我们要运算的数据，或将运算好的数据转为要传送的数字 ASCII 码字符串已经变得非常简单容易了。

这里字符串是可定义长度的（用户自定义），一串数字 ASCII 码转换后，一个双整字就可以表达了；一个双整字转换成字符串后，却变为一串字符串代码。编写梯级逻辑如图 8-29 所示，控制器运行后结果可直接在指令参数上观察。正如我们在指令参数中看到的那样，ASCII 码的显示总是加上引号来区分，这样更不容易被误读了。

下面我们摘取一段关于串口读取 ASCII 码数据梯级逻辑处理的实例，串口连接的是一台条形码扫描仪，其读到的数据是 ASCII 码字符串的表达形式，要将这个读入的数据转为可运算的数值。编写梯级逻辑如图 8-30 所示。

我们来解读这一段梯级逻辑。首先，ABL 指令测试数据缓存区，一旦找到数据，其找到位 FD 置位，为避免读行指令 ARL 被悬挂太久，占用了 ASCII 队列的空间，令 ABL 指令的找到位作为 ARL 指令执行的梯级条件，读行指令 ARL 读取缓存区字符串数据，遇终止符

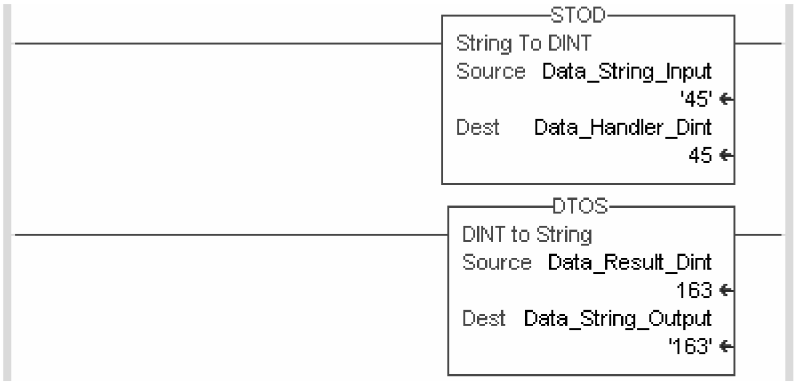


图 8-29　ASCⅡ码和数字的互转

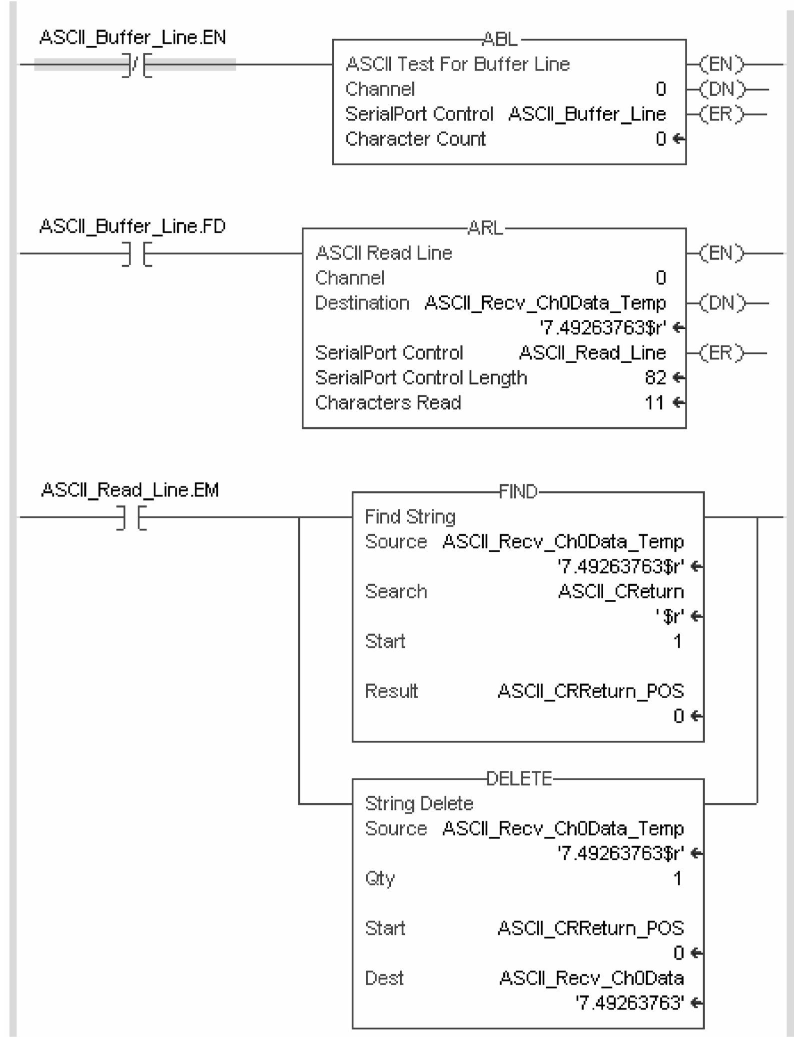


图 8-30　ASCⅡ码转为数字的处理

$r 而停止（终止符在控制器串口组态中设置），从而获取字符串‘7.49263763$r’，读行指令 ARL 空置位 ASCII_Read_Line.EM 的状态作为下一步操作的梯级条件，下一步使用查找字符串指令 FIND 来找到所指定的内容，本条指令指定的是字符串中的终止符，位于位置 0，然后用删除指令 DELETE 删除指定的内容，最后使用 ASCII 码转换数据的指令 STOR 将其转换为实数，该实数被送到模拟量输出模块结构数据的通道 0 数据，并通过 D/A 转换送到输出模块的信号通道。

经过以上梯级逻辑的处理，一个来自于 ASCII 码设备的 ASCII 码字符串通过串口进入控制器，转换为可运算的数据，最终作为一个模拟量输出模块的控制信号送出，实施了外部设备的控制。

再举一个关于 ASCII 数据处理的实例，假定要将控制器系统的一段日期时间记录数据传送到连接在控制器串口的外部 ASCII 码设备中，为完成日期时间记录数组的转换和传送的操作，编写梯级逻辑如图 8-31 所示，这里使用的是数据转为字符串的 DTOS 指令。

首先，例程扫描之初清除 ASCII 码缓冲区和日期时间数组的指针。

将日期时间数组指针所指单元的数据复制到缓存区 DateTime_Buffer，这是一个通过 GSV 指令从控制器系统读取的日期时间结构数据标签，结构为 7 个双整字，分别是年、月、日、时、分、秒、毫秒，此处只复制了一个数据标签待处理。

只截取了日期时间数据结构标签的前 6 个数据，转换指令 DTOS 分别将日期时间结构数据标签的子元素年、月、日、时、分、秒的双整字转为 ASCII 码数据。使用字符串连接指令

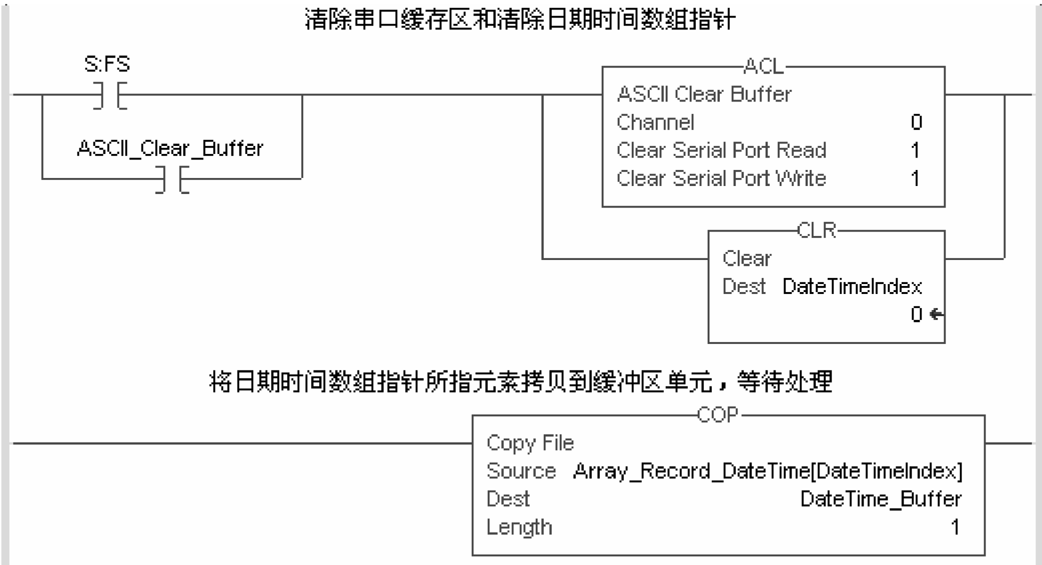


图 8-31　将控制器日期时间转换成 ASCⅡ码

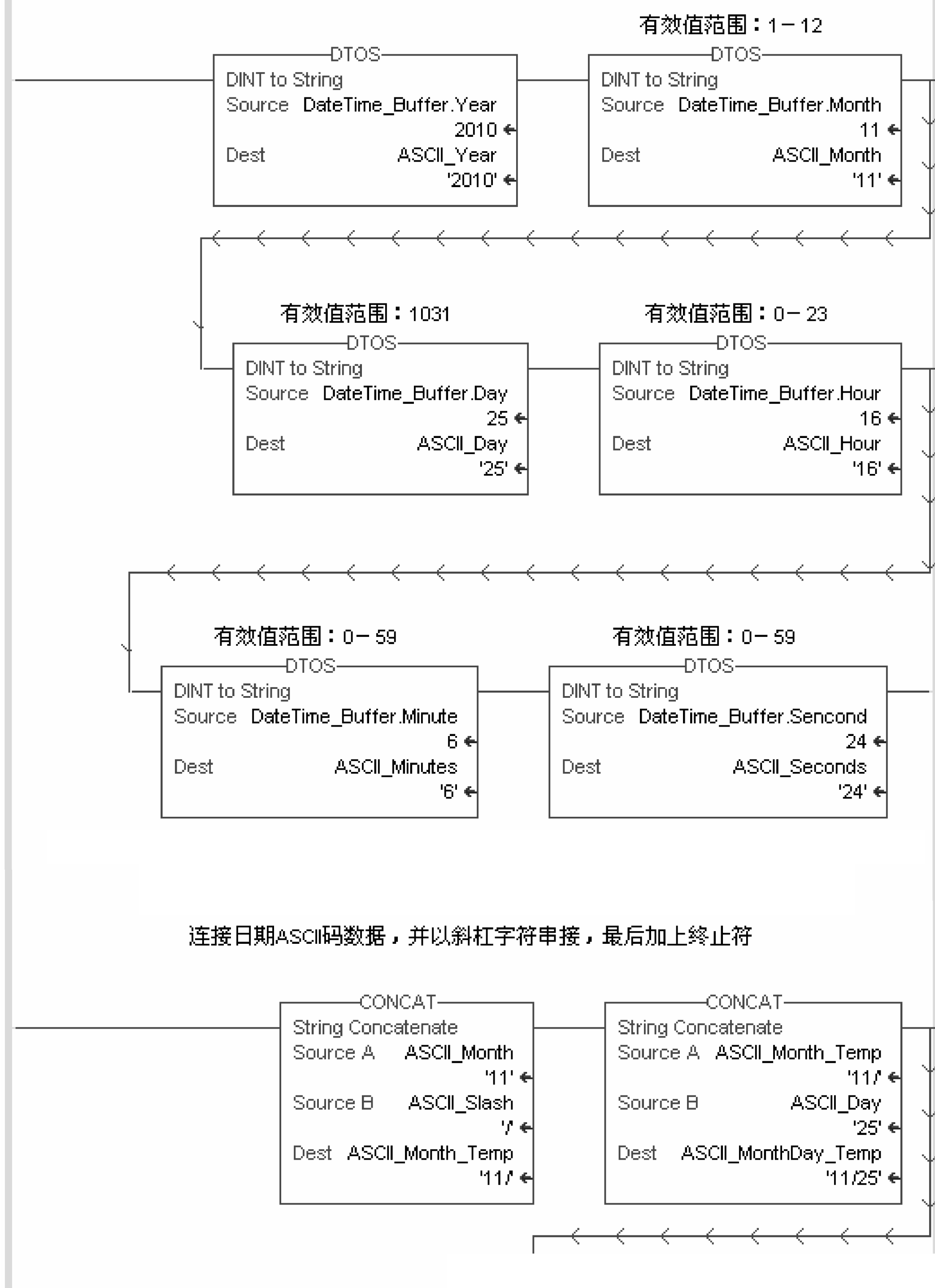


图 8-31　将控制器日期时间转换成 ASCⅡ码（续）

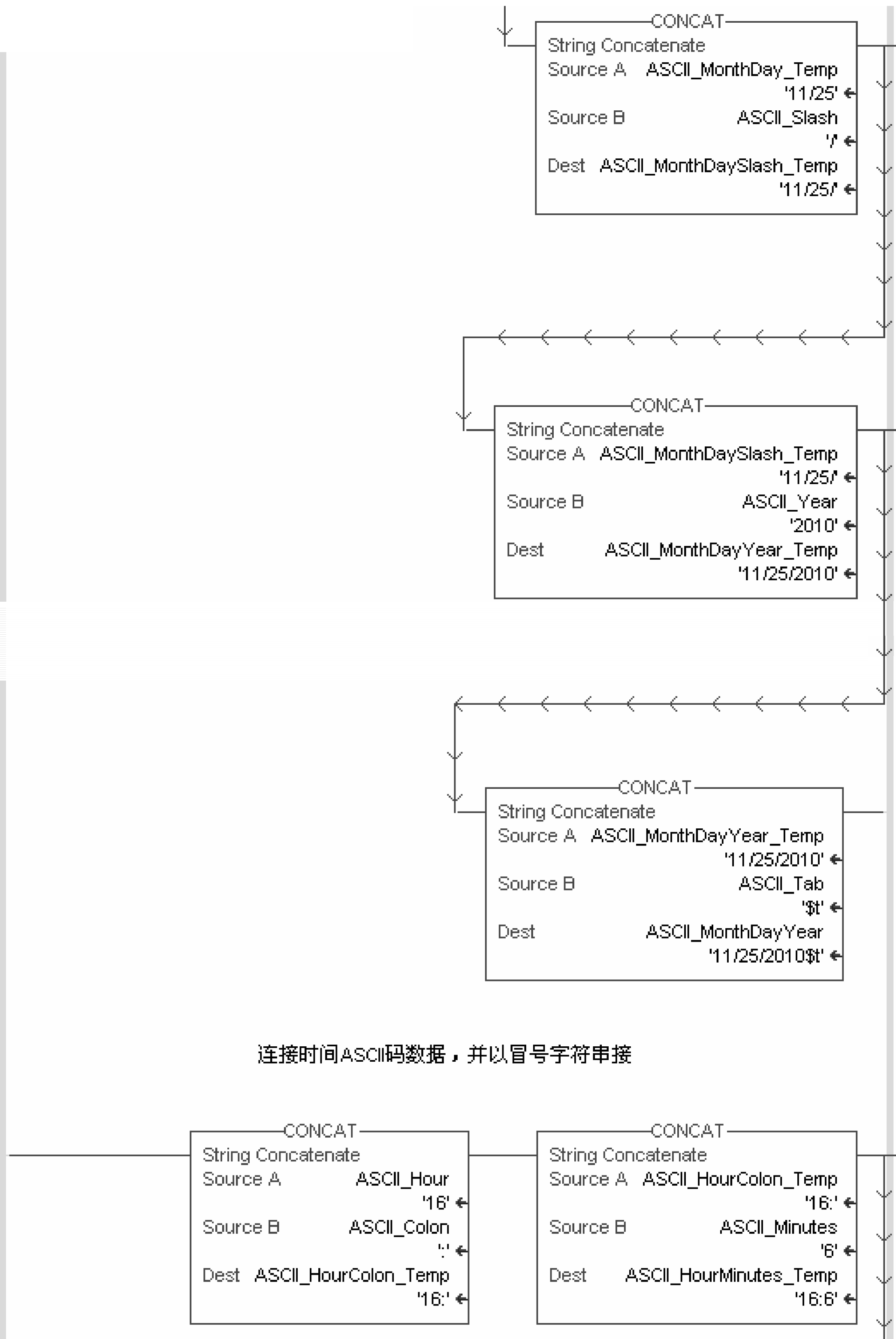


图 8-31　将控制器日期时间转换成 ASCⅡ码（续）

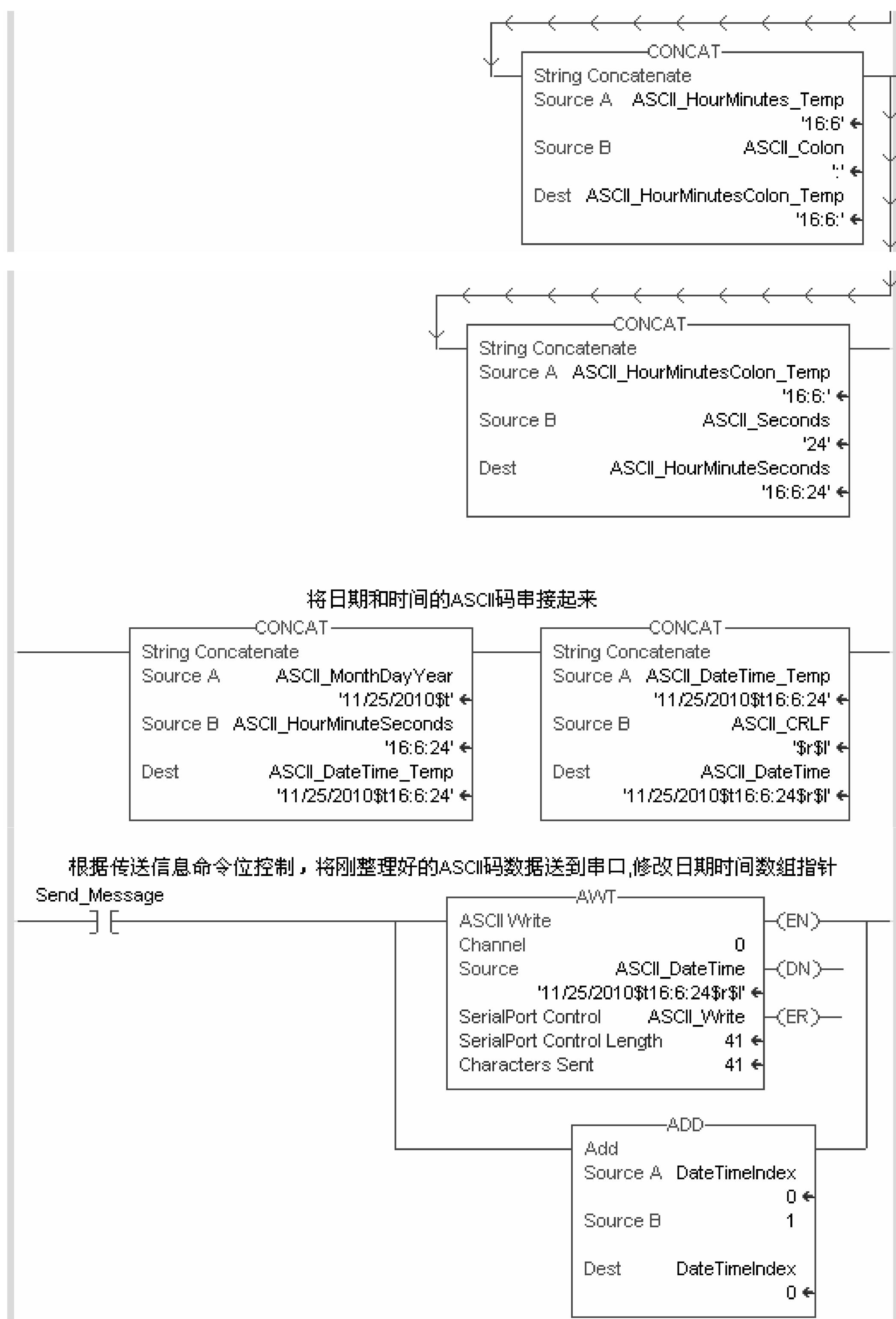


图 8-31　将控制器日期时间转换成 ASCⅡ码（续）

CONCAT 将日期一一连接起来，并加入连接符斜杠，以终止符$t 结尾；再次使用字符串连接指令 CONCAT 将时间一一连接起来，并加入连接符冒号；最后使用 CONCAT 指令将日期和时间的 ASCII 码字符串接起来。

将处理好的字符串用字符串写指令 AWT 送到控制器串口，与此同时，修改日期时间数组的指针准备转换下一个日期时间数组元素。此段梯级逻辑为一条日期时间记录的处理，一批日期时间的处理，还需编写其他的梯级逻辑予以处理。如果采用 Add_On 指令来处理，不失为一个简洁快速的办法，后续的 Add_On 指令的章节中我们再讨论。

第 9 章

数组指令的编程

尽管 PLC 的起步是替代硬件与、或、非门的逻辑电路和硬件的计时器，用软逻辑和计时器指令完成时序的逻辑控制，但是发展到较高阶段后，已不满足于简单的逻辑关系控制，而是希望提高数据处理能力，能够对批量数据进行快速集中的处理，这就是当时所说的文件操作指令。顺序控制功能指令系统已经非常成熟的 PLC5 处理器已具有相当强的数据文件处理能力，众所周知，Logix 控制器的时序逻辑控制部分的指令完全是从增强型 PLC5 的指令系统移植过来的，PLC5 指令系统称为文件操作指令，在 Logix 控制器中通称为数组操作指令，不同的是 Logix 控制器的数组最多可达三维，PLC5 的文件只相当于一维数组，从某些指令的处理能力看，Logix 的多维数组具有更强的批量数据处理能力。

数组操作指令如下所列：

- FAL 数组算术逻辑运算指令；
- FSC 数组搜索比较指令；
- AVE 数组平均值计算指令；
- SRT 数组排序指令；
- STD 数组标准偏差计算指令；
- SIZE 查找元素长度指令。

数组操作的指令尽管用途各不相同，执行过程也有区别，但操作模式却不外乎如图 9-1 所示的三种，数组到数组的操作适合一次性批量数据的处理，通常会选用一次扫描全部执行完毕；元素到数组的操作通常是装入的操作，适合逐个数据处理，一般由梯级条件触发而引起操作；数组到元素的操作通常是卸载的操作，适合逐个数据处理，一般由梯级条件触发而引起操作。元素到数组或数组到元素的操作多见于堆栈指令和顺序控制指令，执行过程的有序与控制过程紧密相关，这些将在专门的章节进行讨论。

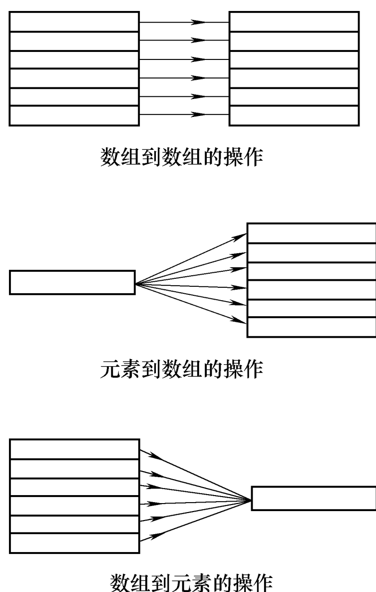


图 9-1　三种操作模式

数组操作指令的操作模式的体现，在 FAL 指令中最为全面，指令参数项 Mode 中操作模式可选，应用了这三种操作方式，其他数组操作指令总也脱离不了其中的一种操作模式，而且大都与梯级条件紧密相关，在编写梯级逻辑时，请留意为这些指令执行提供的梯级条件和

相应的操作模式。

一般地，数组操作指令参数中还需要配备一个控制结构数据标签，这个本质上是计数器的控制结构数据标签用来为数组操作过程计数。在控制结构数据标签中，不但设定数组操作的长度，记录数组操作的当前位置（也称为指针），还需要一些状态位来表达数组操作指令当前的执行状态，控制结构数据如图 9-2 所示。

Name	Value	Style	Data Type	Description
Control	{...}		CONTROL	
Control.LEN	0	Decimal	DINT	长度
Control.POS	0	Decimal	DINT	指针
Control.EN	0	Decimal	BOOL	使能位
Control.EU	0	Decimal	BOOL	卸载使能
Control.DN	0	Decimal	BOOL	完成位
Control.EM	0	Decimal	BOOL	栈空位
Control.ER	0	Decimal	BOOL	出错位
Control.UL	0	Decimal	BOOL	卸载位
Control.IN	0	Decimal	BOOL	禁止位
Control.FD	0	Decimal	BOOL	找到位

图 9-2　控制结构数据

控制结构数据子元素说明：

- LEN（数组操作长度）与数组首个元素一起，指定了数组的操作范围，通常由数组操作指令参数设定。
- POS（指针）指出了数组当前的操作位置，在字对数组操作或数组对字的操作时，尤为重要，通常是数组操作指令使能引起的增加。
- EN（使能位）数组操作指令由于梯级条件的满足被使能时置位。
- EU（卸载使能位）堆栈指令卸载操作使能时被置位。
- DN（完成位）指令的指针等于长度时置位，通常表示到达或完成。
- EM（栈空位）堆栈指令用到的表示，即指针为 0 的状态，此时堆栈数组中没有数据存放。
- ER（出错位）大多数指令当 POS < 0、LEN < 0、POS > LEN 时，出错位置位，个别指令则有不同含义。出错位置位时指令不再执行。
- UL（卸载位）用于移位指令的卸出位装载，装载左移指令或右移指令移出的位状态。
- IN（禁止位）当数组的操作获得探测结果，此位禁止继续探测，直到解除禁止后，数组方可继续探测。
- FD（找到位）搜索等数组操作指令使用，找到符合条件的对象时置位。有时与禁止位配合使用。

以上所列举的是控制结构数据的状态位，不是每一条数组指令都用到了所有的状态位，不同的数组操作指令，使用了不同的状态位，状态位的含义也不尽相同，这需要在每条指令

的运用中加以解释，这里只作一般性的罗列和介绍。

特别要注意的是，数组操作指令只对内部数据进行批量处理，它所用到的控制数据结构、状态都是比较简单的。对外部设备操作的数据块，如 MSG 指令，虽然大都也是批量数据的通信处理，但不会使用控制结构数据来描述它们的操作过程，而是有 MSG 指令专用的更为复杂的结构数据，其中含有的信息十分丰富，不但要指定双方的通信标签，本机与外部设备的通信路径，还有实时的设备通信状态和数据交换状态。

9.1　数组算术逻辑运算指令 FAL 的编程

在 Logix 控制器的指令系统中，虽然我们改口称文件操作指令为数组操作指令，但这条指令的名称却保留了原貌（File Arithmetic and Logic），这是一条用于算术逻辑的批量数据处理的指令，这个批量数据原来称为文件，现在称为数组，且扩展为多维数组。Logix 控制器的 FAL 指令的处理跟 PLC5 指令系统中的 FAL 指令有差别，可以说变得更为灵活，能够选择更多的表达形式，但编写指令参数也变得更为繁杂。

这条指令是一条处理功能很强的指令，它不仅仅完成数组的算术逻辑运算，还可以完成数组的转换和传送。尤其是在数据传送灵活性方面更优于 COP 指令。它可以根据需要对结构数据数组中的单个子元素完成数据连续存储的操作，甚至可以按照一定的规律抽取结构数据中的子元素完成连续存储，这正是 COP 指令无法解决的问题。FAL 指令的处理方式灵活多变，它的执行过程有三种操作模式可选择：

- 整体模式　梯级条件跳变激活指令后，指令在一次扫描中完成所有的操作，一般用于数组对数组的操作。Mode 项选择 ALL。
- 数量模式　梯级条件跳变激活指令后，指令分几次扫描完成所有的操作，适合大批量数据的操作，将操作均衡地分配在几个扫描周期，避免不合理的某次扫描周期过长。倘若操作尚未结束，梯级条件消失，指令将继续执行直到完成。注意，在数组操作完成位没有置位之前，不要使用目标标签中的数据，否则将使用新旧混杂的数据，带来不可预料的结果。Mode 项键入正整数，作为每次扫描的操作个数。
- 增量模式　梯级条件每跳变一次执行一次，一般用于数组对元素的操作或元素对数组的操作。Mode 项选择 INC。

FAL 指令属于保持型指令，指令的执行必须有梯级条件的跳变触发，受 COP 指令的影响，常常被人忽视这一点，提供的是梯级条件成立，而不给予梯级条件跳变，以至于出现指令只执行一次便不再执行的错误。梯级逻辑编写 FAL 指令如图 9-3 所示，本条 FAL 指令的执行结果是将结构数据标签计时器数组的子元素累加值 ACC 抽取出来，传送到双整字数组中连续存放。请注意，如果该条指令的梯级条件 To_ACC 置位后没有复位，这个数据传送动作只会执行一次。

具有操作模式可选的 FAL 指令跟其他数组指令的操作不同，甚至跟 PLC5 的 FAL 指令的操作也不同，主要表现在控制器系统不会令操作对象的数组元素序号顺序地自动增加，而是独立于指令操作之外，交由外部修改，即在 FAL 指令之外编写修改数组元素序号的梯级逻辑，这样可以使得数据批量处理的方式更为灵活。在 FAL 指令之外编写梯级逻辑修改被操作数组元素序号，对于增量模式是可行的，可以在修改好数组元素序号后，按照条件再进行

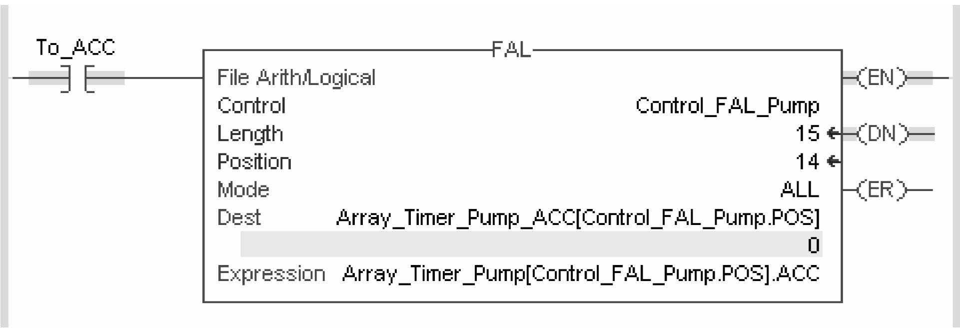


图 9-3 抽取计时器子元素 ACC 送至双整数组中存放

下一步操作，如果 FAL 指令选择了整体模式，是没有机会由外部梯级逻辑的执行来修改数组元素序号的。通常采用间接寻址来获得数组元素序号的移动，如果借助指令本身的控制结构数据标签的指针变化来作为处理对象的数组元素序号，可以在数组元素项中直接填写本条指令控制结构数据的指针，随着指针的增加，元素寻址实现了顺序移动。

间接寻址亦可安排出某种规律。如果数组每隔一个元素的抽取，比如抽取偶数元素的 ACC 值，则可以编写如图 9-4 所示的梯级逻辑，在数组元素寻址项的编写上，只要将控制结构数据标签的指针乘上 2 就可以了。同理，如果希望数组元素序号从 2 开始，则间接寻址令控制结构数据标签的指针加上偏移量 2 即可，其余类推。

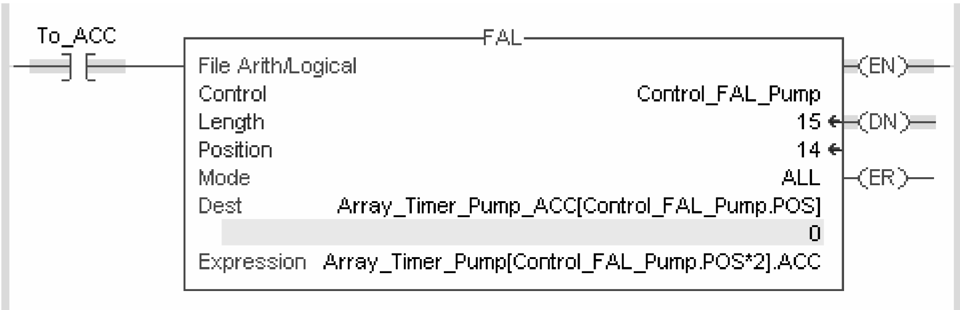


图 9-4 具有特定规律的间接寻址

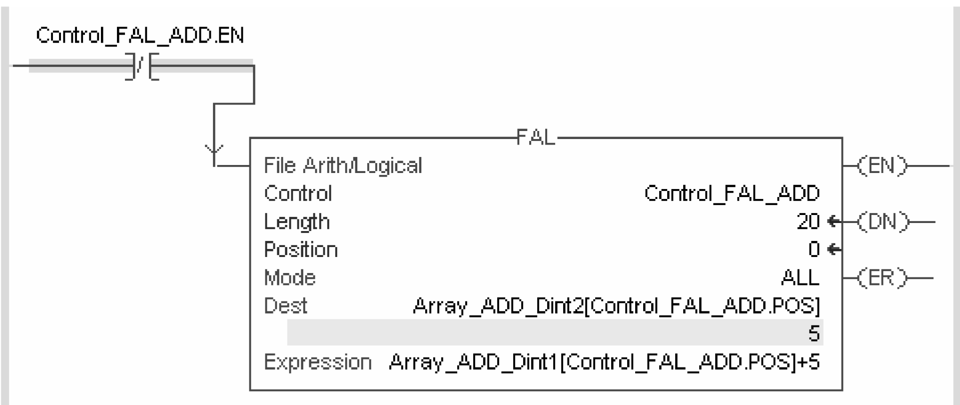


图 9-5 带运算表达式的数组操作

FAL 指令最基本的功能就是数组对数组的运算，编写梯级逻辑如图 9-5 所示。将数组 Array_ADD_Dint1 每个元素加上 5，送到目标数组 Array_ADD_Dint2 的对应元素，由于使用了相同的间接寻址作为元素序号，源数组和目标数组的元素序号是一致的，有效算术连接符加号“+”连接常数 5 即为表达式。梯级条件采用本条指令的控制结构数据标签的使能位常开输入位指令，则可以让这条 FAL 指令反复地执行，这也是让指令反复执行常用的手法，对于需要梯级条件触发才能执行的保持型指令，则能保持其连续不断地执行指令。

FAL 指令的操作可以更为复杂，如用单一操作的转换指令与表达式复合在一起实现运算兼转换的执行动作，FAL 指令也可以实现数组对元素的操作或元素对数组的操作，但要采用增量模式。下面的实例综合了这两种运用，这个实例是一个将数字转换为 BCD 码送到数码显示 LED 的输出；和一个从拨盘输入 TW 读来的 BCD 码转换为数字并存入数组，编写梯级逻辑如图 9-6 所示。

因为两条 FAL 指令选择的操作模式都是增量模式，每当梯级条件 Unload_LED 跳变一

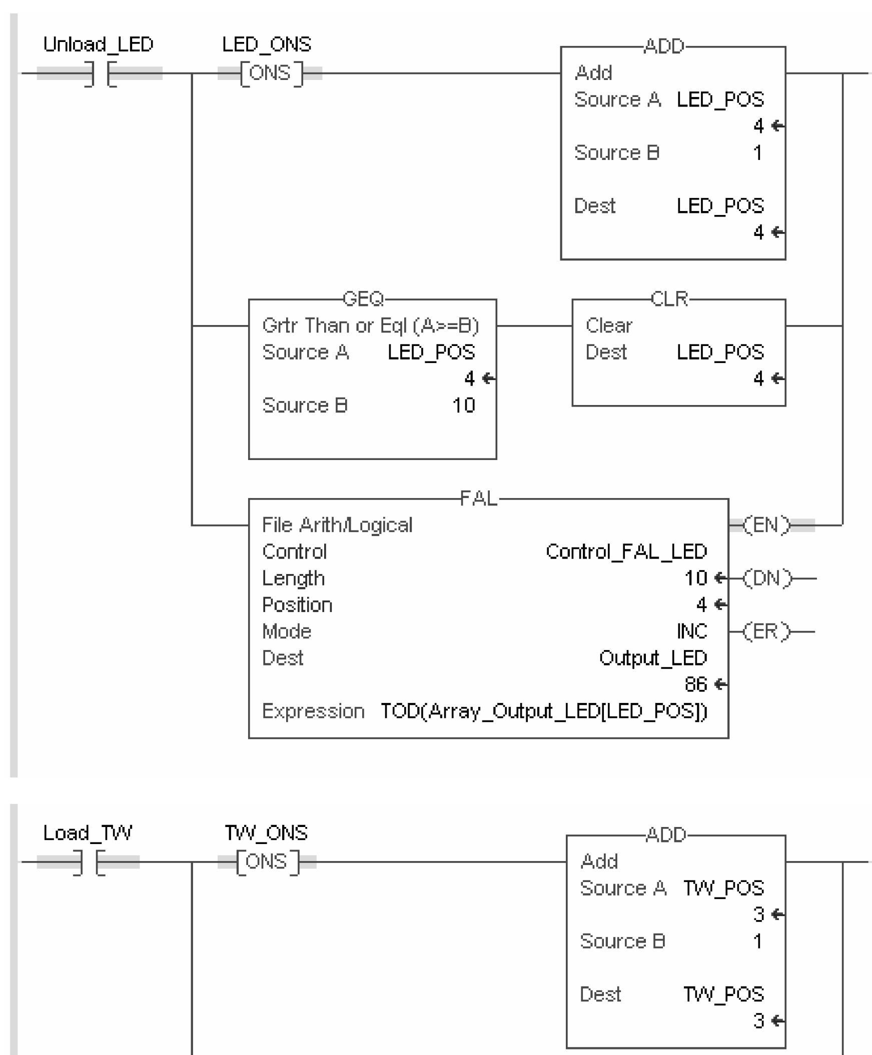


图 9-6　增量模式的数组操作

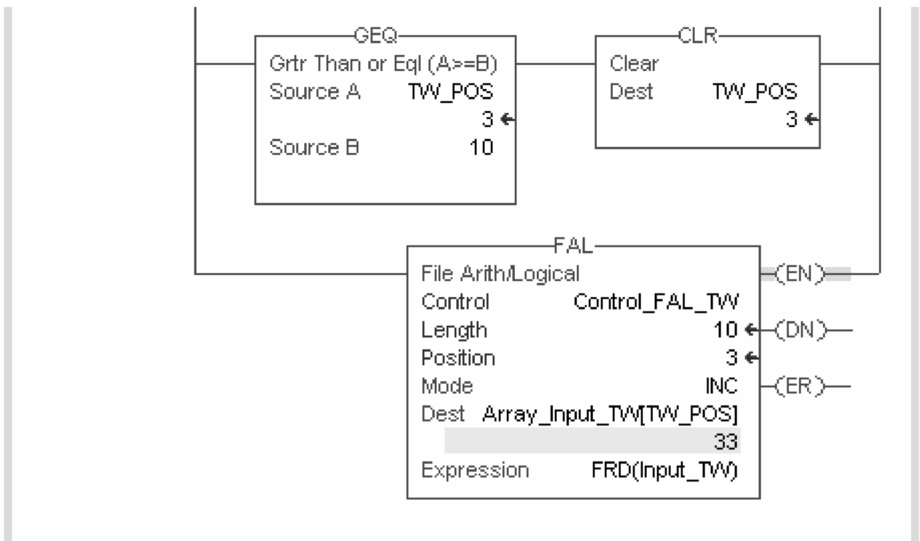


图 9-6　增量模式的数组操作（续）

次，便将指针所指数组元素转换成 BCD 码并送给目标字 Output_LED；每当梯级条件 Load_TW 跳变一次，便将来自 Input_TW 的 BCD 码转换为数字送到指针指向的数组元素。这种操作模式完全类似后面章节将要谈到的顺序器输出指令 SQO 和顺序器装载指令 SQL 的操作。对于数组操作指令来说，这是典型的数组对元素的操作或元素对数组的操作。

外部修改指针可以有任意的形式，方式可以灵活多变，甚至特别指定，这样顺序加 1 的修改，跟直接使用指令结构体的指针做间接寻址是一样的。

试想一下，如果元素和数组的操作不选用增量模式会是什么样的情景，前一条指令是送出，如果不采用增量模式，数组元素不停地输出到 Output_LED，其覆盖的结果是只得到数组最后一个元素的数据，这完全可以用 MOV 指令来得到；后一条指令是读入，如果不采用增量方式，源字的数据将充填整个数组，这未尝不可，但是我们已有一条称为充填指令的 FLL 指令专司其职。

由于采用了增量操作模式，这种逐字的操作完全可以在外部修改数组指针，在 FAL 指令执行之前，需要执行一个指针修改的操作，用加 1 的操作便可实现。但是 ADD 指令自身不能识别是否已经完成一个循环，不会自动回归起始指针 0，采用判断指令来完成，这里采用的不是等于 10 的判断而是大于等于 10 的判断，一旦错过了 10 ，还是能够回到原点。此外，ADD 的操作被 ONS 指令限于只能操作一次。

请注意，外部指针修改的梯级逻辑必须在 FAL 指令执行之前执行，梯级条件跳变不但引起指针的修改，还引发了 FAL 指令的操作动作，在这个操作动作之前必须准备好指针，如果不在意这两者的执行顺序，将 FAL 指令并列在输出指令的最前面，那么结果就完全不会是我们所想要的了。

总而言之，FAL 指令是功能十分强大的指令，只是一条指令就可以完成需要大量梯级逻辑才能完成的任务，它不仅仅是数组的操作，还是复杂运算关系的集合，最有意义的是具有检索的功能，这些特性集合在一起，的确能解决许多棘手的问题，而表达又是如此的简洁和直观。

9.2　搜索指令 FSC 的编程

搜索指令 FSC 的操作跟 FAL 指令相似，可以选择操作模式，选择整体模式（ALL）时，梯级条件变为真开始执行所有的搜索，直到找到符合条件的元素为止。选择增量模式（INC）时，梯级条件跳变一次，指令执行一次，不管是否找到符合条件的元素都会停止。当指令执行的时候，对表达式中建立的比较关系和比较对象进行比较，找到满足比较关系的元素时，找到位 FD 置位，POS 则指出找到的元素所在位置，同时禁止位 IN 也置位，以防止往下搜索，当 IN 和 FD 复位后，继续往下搜索。指令表达式的书写规则与 CMP 指令相同，比较数组和被比较数组的元素序号也是由外部修改的，在整体模式，只能借助于指令自身的控制结构标签的指针来间接获得。

有两组数据进行比较，寻找数据相同的单元，当操作员在人机界面 PanelView Plus 上按下 Start_Search 的操作按钮，比较搜索开始，每当找到相等的数据单元，在人机界面 PanelView Plus 屏幕上显示数组的位置和数据单元的数值，操作员看到后，按下确认键，继续搜索，寻找下一个相同的数据。编写梯级逻辑如图 9-7 所示。

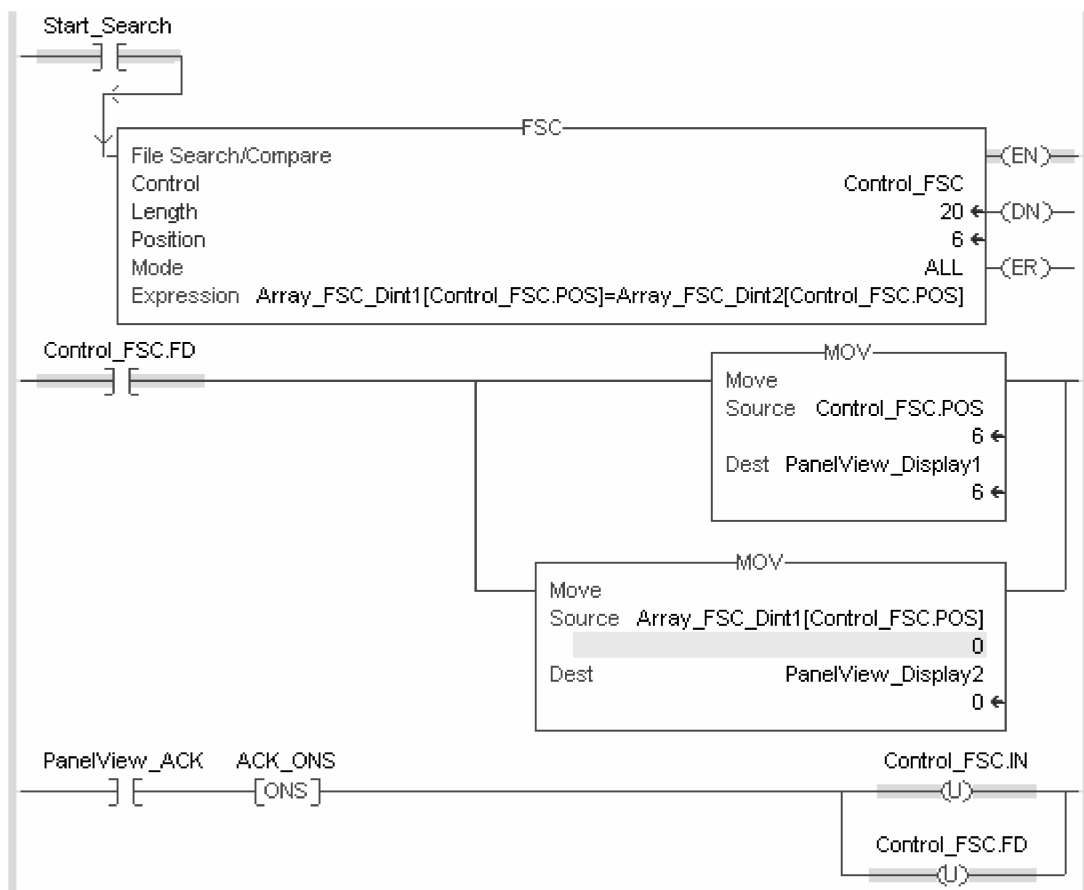


图 9-7　与人机界面配合的搜索

当梯级条件 Start_Search 成立，FSC 指令开始执行，顺序比较数组的每个元素，直到找到符合条件即数值相等的元素时才会停止搜索，并展现搜索的结果，将禁止位 IN 和找到位 FD 复位后才能继续下面的搜索。禁止位 IN 和找到位 FD 的确认复位必须用 ONS 指令来确保只能执行一次。

如果采用增量方式来搜索，编写梯级逻辑如图 9-8 所示，改变操作模式的设定。

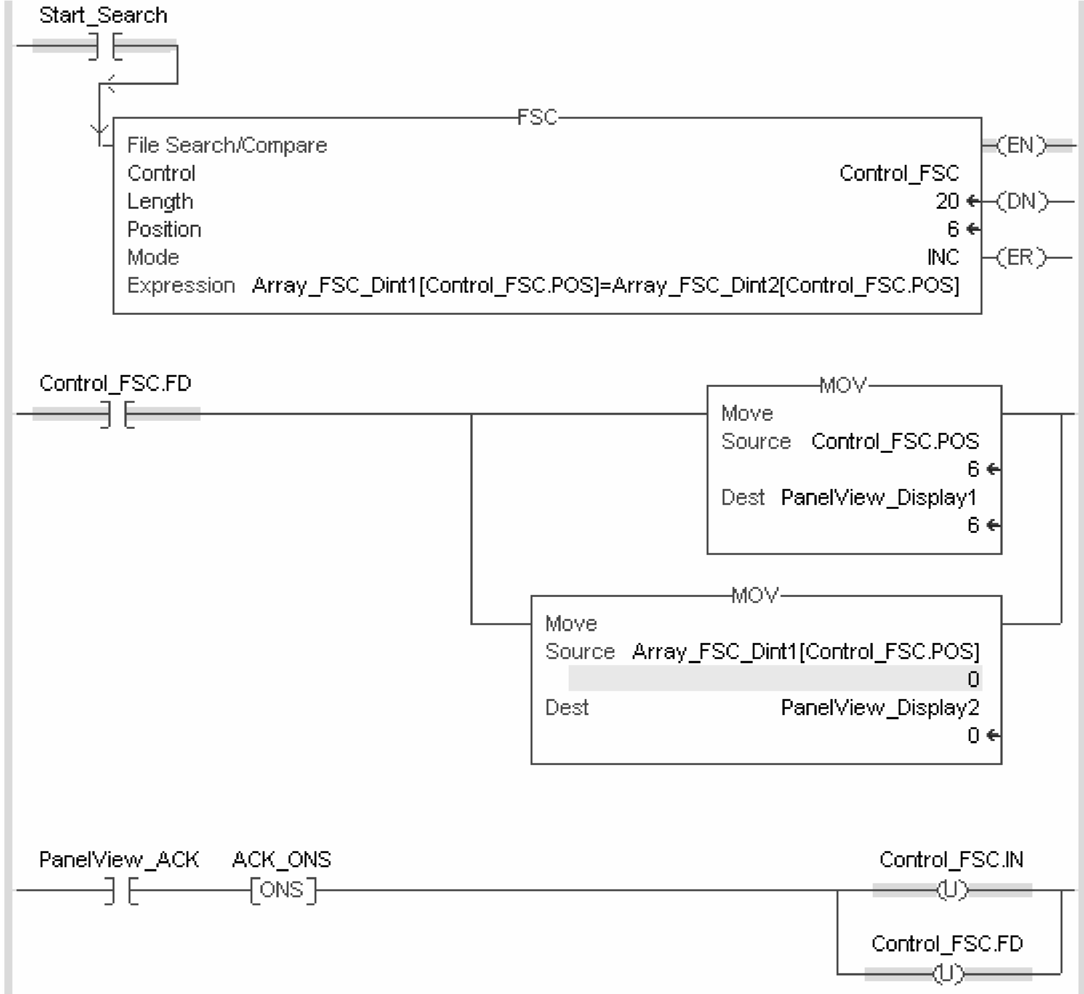


图 9-8 编写的梯级逻辑

试着像 FAL 指令那样，用外部指针修改的方式，结果是失败的，原来一旦找到位置位，无论梯级条件如何跳变，FSC 指令的指针再也不会增加，直到确认位复位才能继续。而外部指针的修改是不受此约束的，这使搜索功能失去了意义，所以 FSC 指令还是要依赖内部修改。

9.3 数组平均值计算指令 AVE 的编程

平均值计算指令 AVE 是一条保持型输出指令，梯级条件跳变触发执行。数组平均值计

算指令 AVE 可以计算指定数组的一维、二维或三维元素的平均值。数组的参数项应填写被计算数组的第一个元素，指定数组的维数可以是 0、1 或 2，长度的参数项应填写参加计算的元素的个数，计算结果必为双整数或实数。本条指令的操作将影响到算术标志位。梯级逻辑编写如图 9-9 所示。

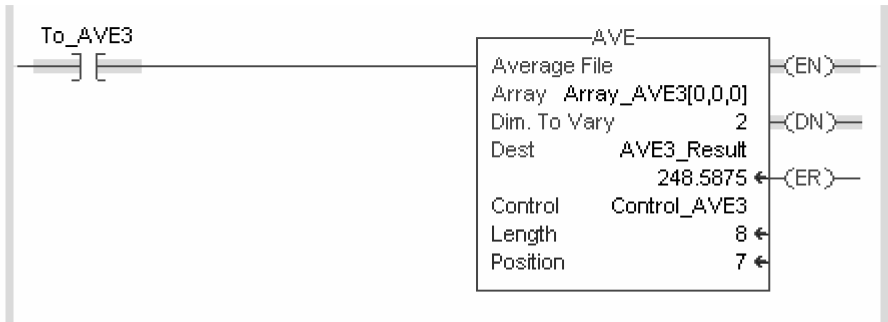


图 9-9　数组平均值计算操作

当梯级条件跳变触发时，指令执行将实数三维数组 Array_AVE3 的 8 个元素求取平均值，并将结果放在实数目标标签 AVE3_REAL 中。指令执行后的数据表中的数据显示如图 9-10 所示。

− Array_AVE3	{...}	Float
Array_AVE3[0,0,0]	234.5	Float
Array_AVE3[0,0,1]	256.7	Float
Array_AVE3[0,1,0]	264.3	Float
Array_AVE3[0,1,1]	253.2	Float
Array_AVE3[1,0,0]	247.6	Float
Array_AVE3[1,0,1]	238.1	Float
Array_AVE3[1,1,0]	242.6	Float
Array_AVE3[1,1,1]	251.7	Float
AVE3_Result	248.5875	Float

图 9-10　AVE 指令执行后的数据表中的数据显示

有一个求取加权平均值的实例，要对模拟量输入模块 3 通道的模拟量数据定时采集，当前采集的数据值与前 19 次采集的数据值累加之后，求取其平均值，编写梯级逻辑如图 9-11 所示。

采集 20 个数据并缓存起来需采用堆栈指令，关于堆栈指令 FFL 和 FFU 的执行过程后续章节将详细介绍。连续采集的 20 个数据值存放在数组 Array_AVE 中，这里将对连续采集到的 20 个数据值求取平均值，设定计时器 Timer_AVE 预置值为 1s，计时器的完成位 DN 作为工作位，则每秒钟采集数据一次。

指令 AVE 的梯级条件限制了求平均值的运算，在数据未采满 20 个时，不进行计算，当 20 个数据采集满之后，每采集一个数据计算一次平均值，用于控制数据采集的计时器的完成位同时提供了一个跳变的梯级条件，使得 AVE 指令在每次数据采集时得以执行。AVE 指令的计算结果可提供给其他梯级逻辑的指令使用，或者作为某个变化数据的预处理。

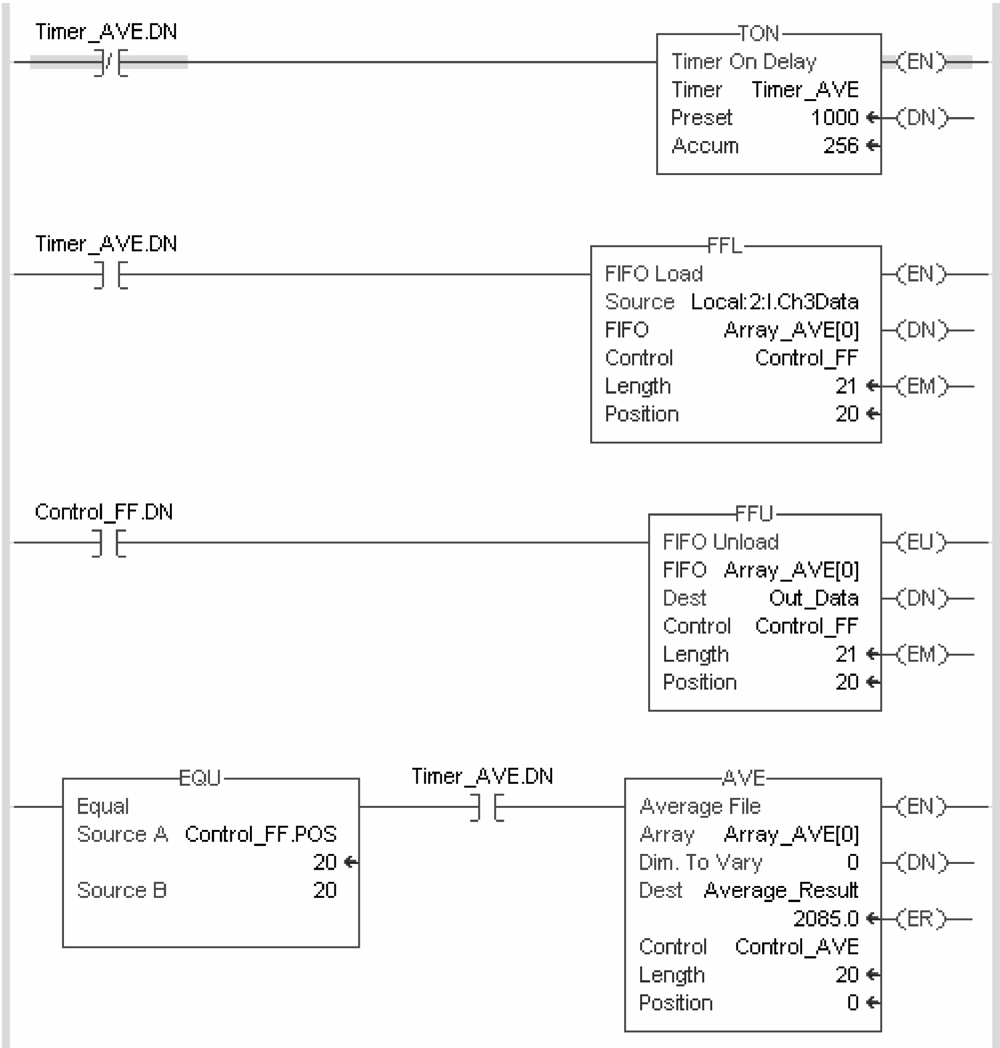


图 9-11 求加权平均值梯级逻辑

这里，堆栈数组的控制结构标签的 POS 应该在初始化例程中给予清除，否则例程运行最初的计算数据因新旧数据混杂而不可靠。

9.4 数组排序指令 SRT 的编程

数组排序指令 SRT 是保持型输出指令，梯级条件跳变触发执行。数组排序指令对指定数组内的一维、二维或三维数组元素的数值从小到大进行递增的排序，数组的参数应填写被排序元素组的第一个元素，指定数组的维数可以是 0、1 或 2，长度的参数应填写参加排序的元素的个数，如果长度超出操作数组的范围，将发生不可预料的后果。此指令依靠梯级条件的跳变触发执行，梯级条件每触发一次指令执行一次。指令的操作将影响到算术标志位。编写的梯级逻辑如图 9-12 所示。

本条指令在梯级条件跳变时执行，将二维数组 Array_SRT 的 12 个元素按升序排序，图

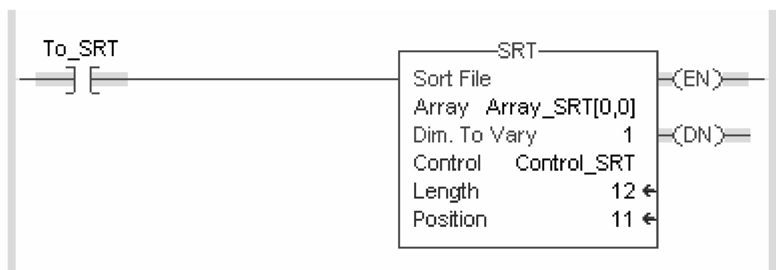


图 9-12　执行数组排序的梯级逻辑

9-13 所示是排序前后数据表的状态。

排序前

Name	Value
Array_SRT[0,0]	123
Array_SRT[0,1]	234
Array_SRT[0,2]	187
Array_SRT[0,3]	398
Array_SRT[1,0]	247
Array_SRT[1,1]	166
Array_SRT[1,2]	267
Array_SRT[1,3]	89
Array_SRT[2,0]	25
Array_SRT[2,1]	182
Array_SRT[2,2]	289
Array_SRT[2,3]	456

排序后

Name	Value
Array_SRT[0,0]	25
Array_SRT[0,1]	89
Array_SRT[0,2]	123
Array_SRT[0,3]	166
Array_SRT[1,0]	182
Array_SRT[1,1]	187
Array_SRT[1,2]	234
Array_SRT[1,3]	247
Array_SRT[2,0]	267
Array_SRT[2,1]	289
Array_SRT[2,2]	398
Array_SRT[2,3]	456

图 9-13　排序前后数据表的状态

9.5　查找元素长度指令 SIZE 的编程

查找元素长度指令是非保持型输出指令，梯级条件成立时执行，对指定数组内的一维、二维或三维数组的元素查找元素的个数，指定数组的维数可以是 0、1 或 2，将查找结果放在指令参数 SIZE 的操作数中，指令的操作不影响到算术标志位。编写梯级逻辑如图 9-14 所示。

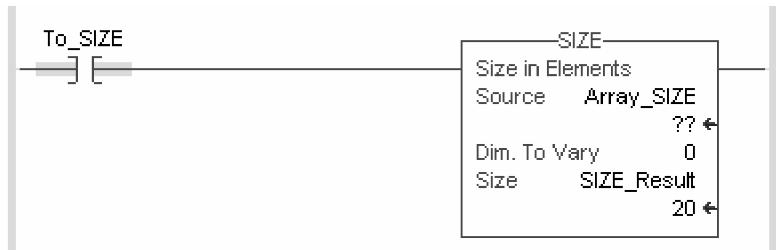


图 9-14　查找元素长度的梯级逻辑

本指令将查找一维数组 Array_SIZE 的元素个数。执行结果为 20，将放在目标标签 SIZE_Result 中。

9.6 数据传送的常规编程

至此，我们已经学习了各种数据传送处理的指令，在用户程序项目中，数据传送是最经常用到的，针对不同的数据类型，选用不同的数据传送指令是颇为讲究的。在所有的控制代码实施执行之前，针对某些数据的预处理工作，通常跟控制对象的控制过程有关，跟现场的设备初始状态有关，跟要执行的初始动作有关。

一般数据传送完成的目的是：

- 外部数据与内部标签的关联；
- 为例程执行代码提供初始数据；
- 清除数据的残留状态。

在控制代码执行之前，将所有的外部输入数据映像给具有设备名称的描述性内部标签，这些内部标签也许是独立操作数标签，也许是针对某个控制对象的用户自定义结构数据的子元素，在例程的执行代码中被引用。映像不但将内外数据关系挂接起来，同时给将要执行的控制代码提供了稳定的数据，消除了例程扫描与 I/O 刷新不同步的隐患。所谓映像就是令这些数据保持一致，用数据传送指令来实现。

有的例程执行代码需要一个初始值，这个初始值可能为 0，也可能是一个指定的数值，有时甚至是一个组态数据块的数据设定。在用户项目中，常常可以见到一个称为初始化的例程，初始化例程是由用户编制的，将项目中所有的初始化处理集中在一个例程中，用特定的关键字 S:FS 作为梯级条件来调用初始化例程。S:FS 只有在例程第一次扫描时置位，之后便处于复位状态，初始化例程被调用一次之后不再执行。

初始化例程中往往含有清零的操作，这是为整个项目运行做好准备，将过去运行的残留情况统统清除。但是在控制器运行期间，有时也需要清除残留状态，特别是当例程的调用从一个状态切换到另一个状态时，刚刚运行过的数据会残留在不确定状态，尤其是像指针一类的数据，更是影响到操作进程，所以在进入一个新的工作状态时，为确保执行的严谨和精准，一定要清除残留状态。

不管数据传送的目的是什么，不同的数据类型要采用不同的数据传送指令来执行，通常来说，数据传送要处理的对象是：

- 位的传递或转换；
- 字的传递或转换；
- 结构数据的传递或转换；
- 数组的传递或转换；
- 结构数组中的某个子元素的抽取并集合。

现在，我们归纳在三种数据传送目的的情形下，针对不同的数据类型使用不同数据传送指令编写的梯级逻辑。

关联外部数据与内部数据的映像执行的梯级逻辑如图 9-15 所示。这些梯级逻辑通常被编写在任务中最先执行的例程里，为后续例程梯级扫描提供稳定的数据，位地址用非保持型

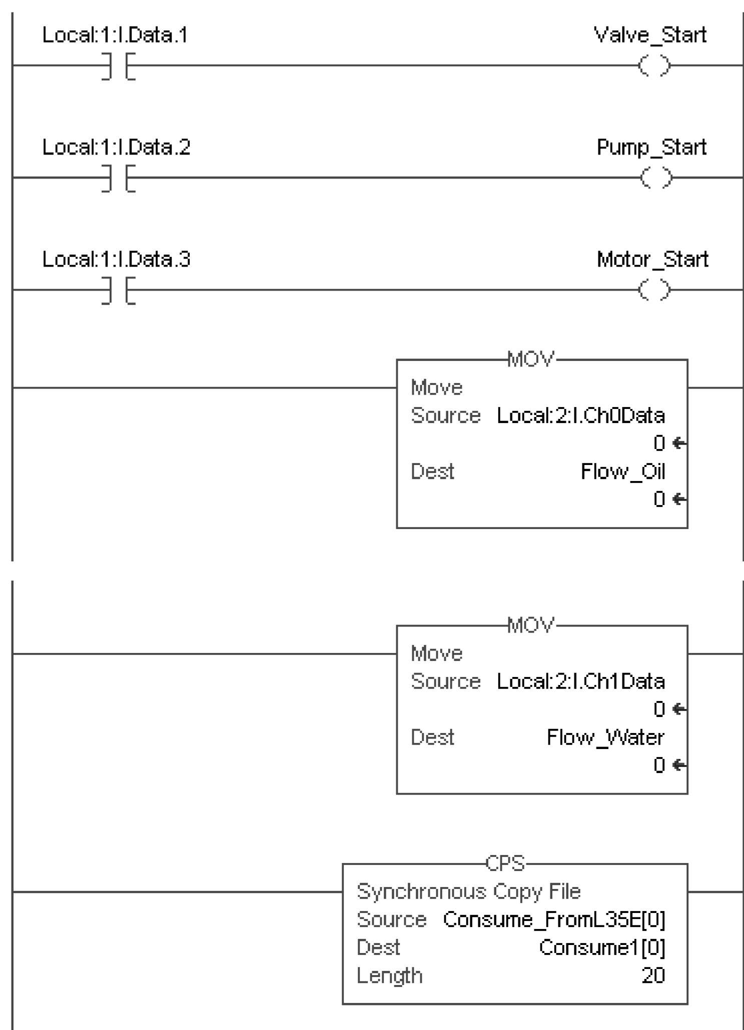


图 9-15　关联外部数据与内部数据映像的梯级逻辑

的位指令映像；字地址用 MOV 指令映像；结构数据用 COP 指令映像，Consume 标签则要考虑新旧数据混杂的风险，采用同步复制指令 CPS，避免外部数据刷新会中断复制过程。

数据初始化执行的梯级逻辑如图 9-16 所示。这些梯级逻辑通常被编写在初始化例程中，只在最初一次扫描时调用，位地址用解锁位指令复位；字地址用清除指令 CLR 复位；数组用充填指令 FLL 复位或者设置；结构数据子元素用 FAL 指令抽取并充 0 复位，控制结构数据指针 POS 和计数器累加值 ACC 均给予充 0 清除。

初始化除了清除残余状态，还可以设定一些初始值，特别是一些组态数据块中的数据，应避免在数据表中直接设置，如果在初始化中用梯级逻辑直接进行设置，一旦出现组态数据混乱，重启运行便可得到解决，是值得推荐的做法。梯级逻辑如图 9-17 和图 9-18 所示。

如图 9-19 所示是初始化例程的调用梯级，首次扫描梯级条件成立，一次性调用初始化例程，所以初始化例程中的梯级几乎都是无条件的，它们只有执行一次的机会。

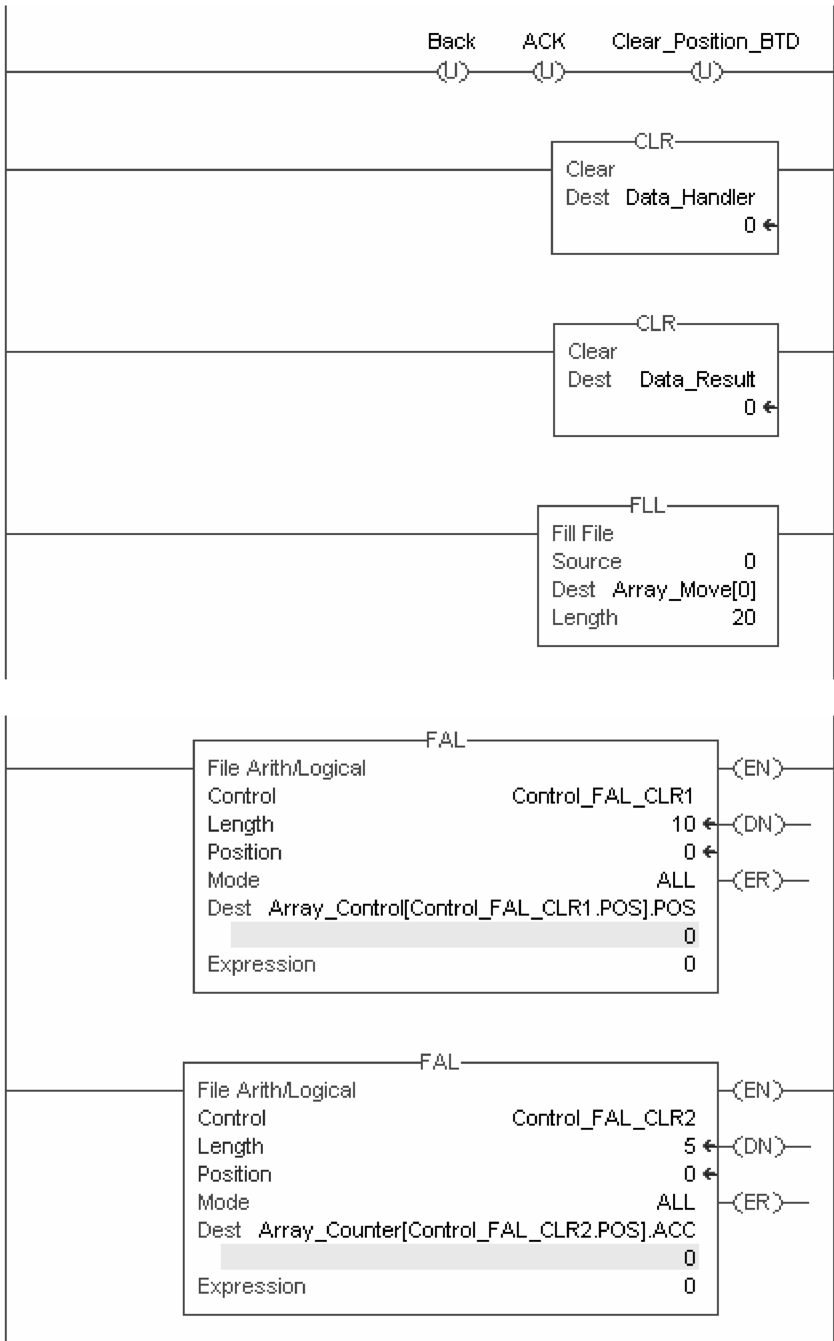


图 9-16　数据初始化的梯级逻辑

除了在特定的例程中对数据进行设定和清除，在某些进程的调用中，同样也需要清除原来的残留状态。如图 9-20 所示是抽取出来的各个嵌入普通例程梯级逻辑中的设置，同样也是用数据传送指令来实现的，MOV 一个立即数 1 给 Buffer_Shift，位移数组获得最初的一个置位状态；FLL 一个立即数 1 给数组 Setp_SFC，让每个 SFC 流程的步都回到初始步；当外部修改的指针到达，判断 POS 大于等于操作上限时，清除指针回到起始值 0。

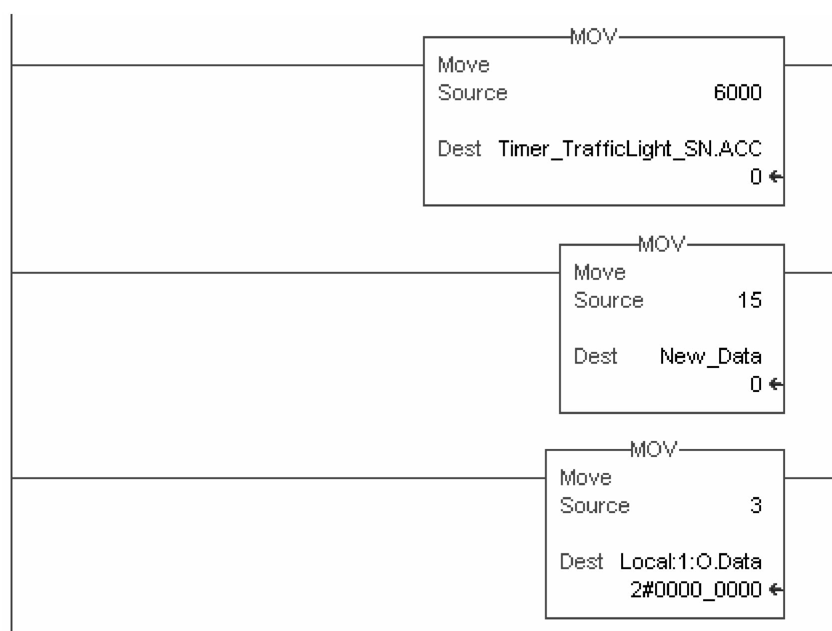


图 9-17　初始值设定的梯级逻辑

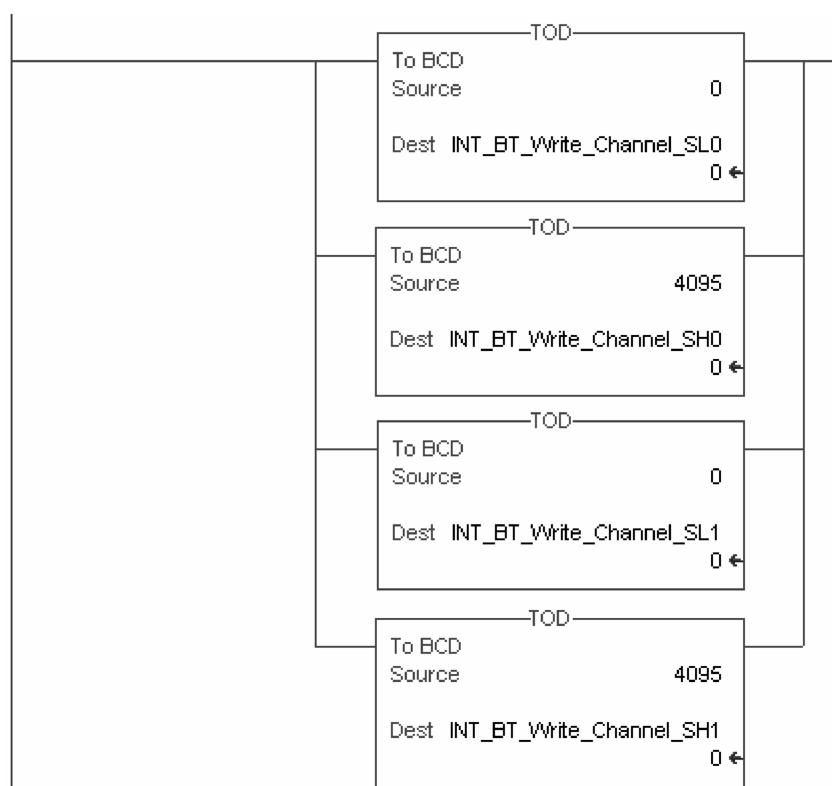


图 9-18　初始值设定的梯级逻辑



图 9-19 初始化例程的调用梯级

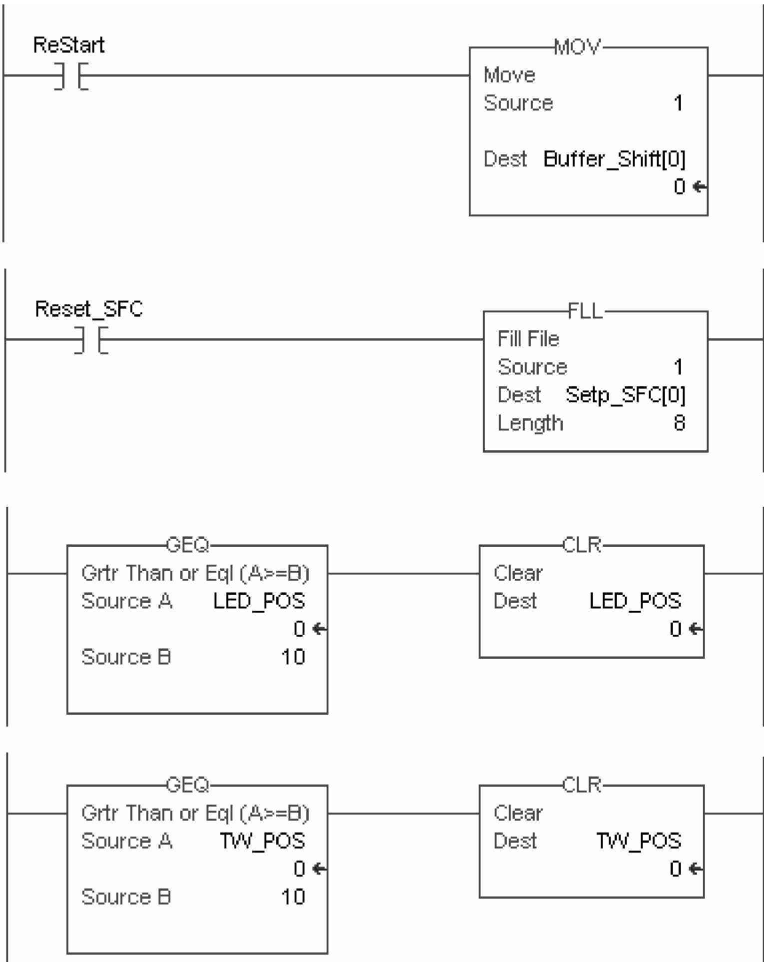


图 9-20 各个嵌入普通例程梯级逻辑中的设置

第 10 章

寄存器指令的编程

在自动控制系统产品软件化如此普及的今天，我们仍然依稀看到硬件系统的痕迹，特别是与寄存器动作紧密相关的一些指令，几乎跟我们以前学习的数字电路中描述的硬件一样，尤其是执行过程，令人强烈地感觉到早期 CPU 汇编语言是如何准确地复现寄存器硬件的动作，源自汇编语言的 PLC 的指令系统自然避免不了寄存器硬件操作的相似。本章要讨论的是有关寄存器指令的操作，这类指令共有某些特征，比如说都需要梯级条件的触发才能执行，正如我们所了解的数字电路中的脉冲触发而引起的硬件动作。前面学习过的计数器指令就是非常典型的寄存器的操作。下面，我们还要继续学习有关寄存器数组操作的指令，寄存器指令如下所列：

- BSL　BSR　位左移指令和位右移指令；
- FFL　FFU　先入先出堆栈指令；
- LFL　LFU　先入后出堆栈指令；
- SQO　顺序器输出指令；
- SQL　顺序器装载指令；
- SQI　顺序器输入指令。

寄存器指令是典型的数组操作指令，其明显的特征是必须分配一个控制结构数据标签给指令，用来记录指令的执行情况，即使是左移指令和右移指令，虽没有指针的表达，也仍然需要控制结构数据标签来记录指令执行状态。寄存器指令还是典型的元素对数组的操作，或数组对元素的操作，下面学习寄存器指令时请留意这些。

10.1　位左移指令 BSL 和位右移指令 BSR 的编程

位左移指令 BSL 和位右移指令 BSR 正是硬件执行特征最明显的寄存器操作指令，我们来看位移指令参数输入的一些规定：

- 移动的操作对象必须是双整数组而不是 BOOL 数组或其他数组；
- 参加移动位范围的第一个位必须是双整字元素的第 0 位；
- 参加移动位范围的最后一位所在的双整字元素的剩余的高位均为无效地址，不能在其他地方引用这些位地址。

这些规定或限制，仅仅是因为位左移指令或位右移指令的寄存器操作都是针对一个寄存器单元来操作的，Logix 控制器的 CPU 是 32 位，它所处理的基本数据单元也就是 32 位，即 32 个寄存器构成的数据单元，对于寄存器移位操作而言，是整个基本数据单元同时移动的，

即每当移动指令被触发，寄存器数据单元所有的位将全体移动。

此外，指令参数的控制结构数据的长度用来指定数组中参加移动位的个数，而不是数组的元素个数，参加移动的位数即使不是 32 位的整倍数，也将耗用整倍数的内存字节，当然剩余位都是无效并不可引用的，因为这些剩余位仍然参加了本寄存器单元的移动，保留着移入的残留状态，显然，这些位状态是无意义的。

位左移指令编写梯级逻辑如图 10-1 所示，每当梯级条件 ShiftL_Pluse 跳变时，触发 BSL 指令执行，移动数组 Array_Bit 参加移动的位向左移动 1 位。位左移指令 BSL 执行移动示意图如图 10-2 所示，移动数组最高位 Array_Bit［1］.25 卸载到 Control_BSL.UL，同时源位标签 Input_L 的状态值补充装载到移动数组的起始位 Array_Bit［0］.0。当移动范围超过一个双整字，前一个双整字的高位会移动到下一个双整字的低位。

这里所提到的寄存器向左移动指的是双整字的低位向高位移动。此处有 58 个位参加移动，不足两个寄存器数据单元，但占用了两个寄存器数据单元，数组中剩余的 Array_Bit［1］的 26～31 位废除，不能用于别的用途。

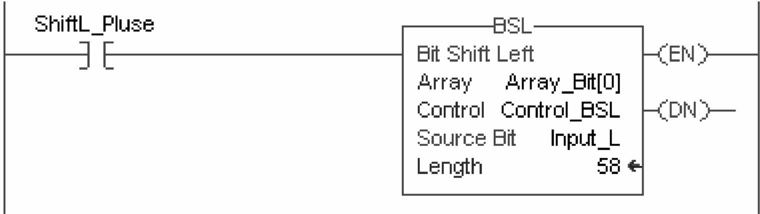


图 10-1　位左移指令梯级逻辑

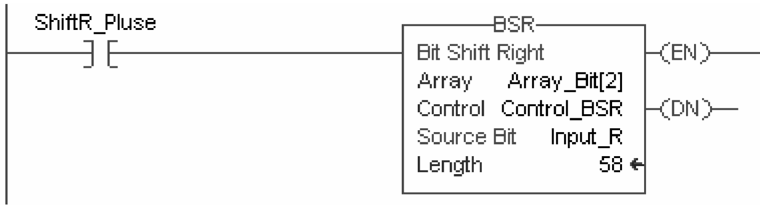


图 10-2　位左移指令 BSL 执行移动示意图

位右移指令编写梯级逻辑如图 10-3 所示，每当梯级条件 ShiftR_Pluse 跳变时，触发 BSR 指令执行，移动数组 Array_Bit 参加移动的位向右移动 1 位。位右移指令 BSR 执行移动示意图如图 10-4 所示，移动数组最低位 Array_Bit［2］.0 卸载到 Control_BSR.UL，同时源位标签 Input_R 的状态值补充装载到移动数组的最高位 Array_Bit［3］.25。当移动范围超过一个双整字，后一个双整字的低位会移动到前一个双整字的高位。

这里所提到的寄存器向右移动指的是双整字的高位向低位移动。此处有 58 个位参加移动，不足两个寄存器单元，但占用了两个寄存器单元，数组中剩余的 Array_Bit［3］的 26～31 位废除，不能用于别的用途。

下面使用 BSL 指令和 BSR 指令共同来完成这样一个需求：令编程实验设备面板上的

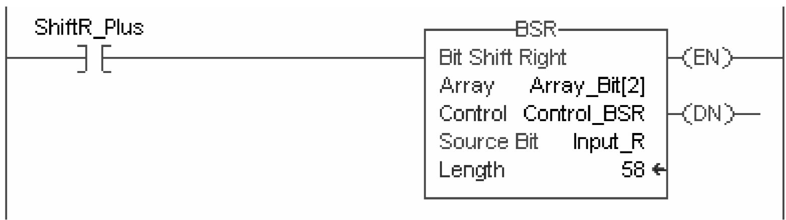


图 10-3　位右移指令梯级逻辑

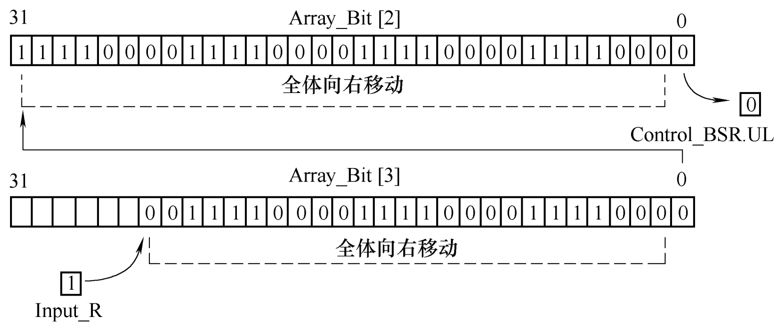


图 10-4　位右移指令 BSR 执行移动示意图

DO/0 灯最初点亮，然后在输出的 4 个灯中，左移 10 次和右移 10 次交替反复执行，移动速度为每秒移动一次，梯级逻辑编写如图 10-5 所示。

正如我们常做的那样，用一个自复位的计时器来产生定时动作，按照要求，每秒要产生一个动作脉冲，计时器预置值设为 1000，每隔 1s 计时器 DN 位置位，保持一个扫描周期。计时器的 DN 位定时脉冲为位左移指令或位右移指令提供了所需的梯级条件触发脉冲，保证

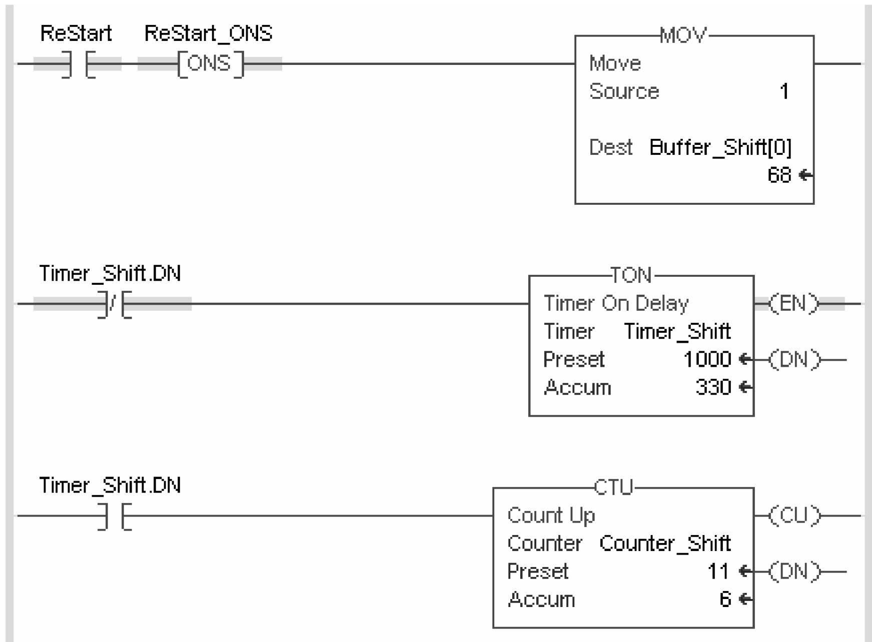


图 10-5　位左移和位右移交替进行的梯级逻辑

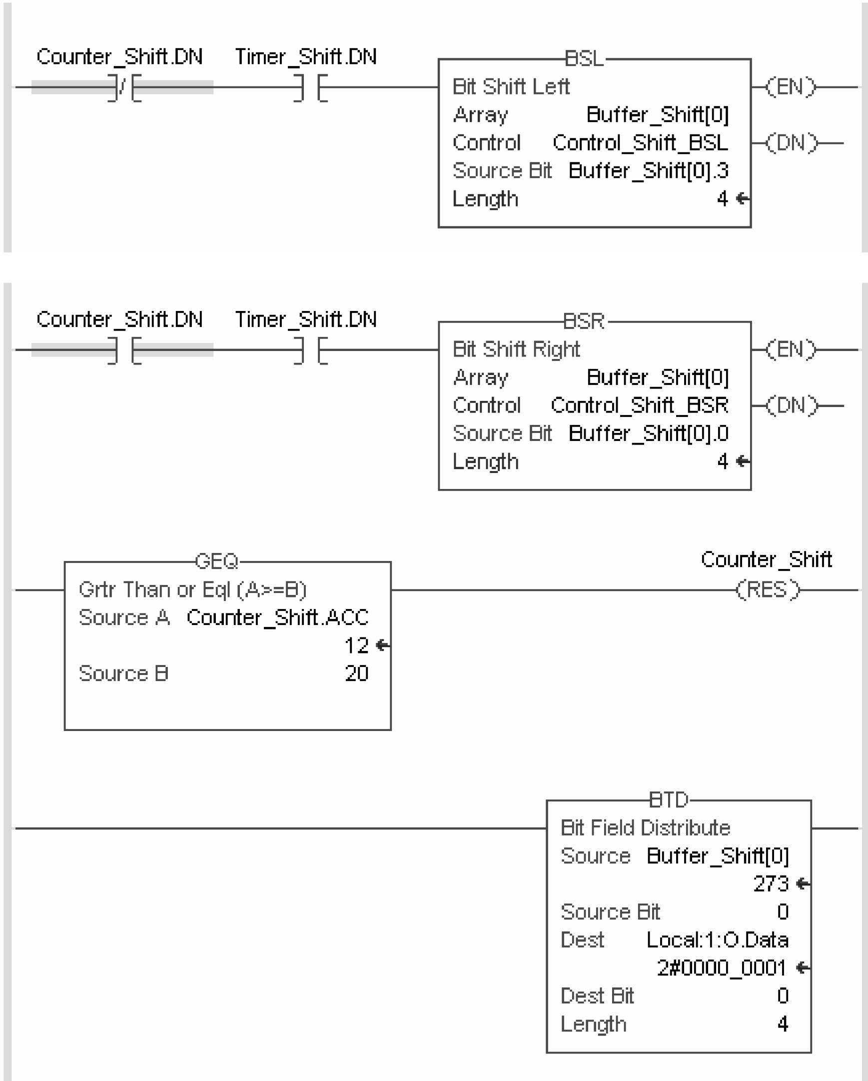


图 10-5　位左移和位右移交替进行的梯级逻辑（续）

了移动对象每秒移动一次。

同时，计时器定时产生的脉冲也为计数器提供了计数脉冲，计数器将作为左移和右移次数的记录。左移和右移各 10 次，可以采用计数器完成位的两种状态来互锁，即计数器 DN 位置位或复位，当计数器计数从 1～10 完成位 DN 呈复位状态，计数从 11～20 完成位 DN 呈置位状态。

这时我们就不用计数器自身的完成位来复位计数器了，而是采用比较指令来复位计数器。注意，这里最好采用大于等于比较条件，不要仅仅用等于比较条件，否则一旦错过，就再也没有复位计数器的机会了。当然还有一种做法是将计数器的预置值设为 20，用 DN 位来复位计数器，这样位左移指令 BSL 和位右移指令 BSR 的前面则要用比较指令来判断梯级条件。如果是多段判断，用计数器作为总循环周期，每个段用比较指令来判断自然是不错的，如果正好只有两段判断，采用计数器的 DN 位就更为简洁了。

注意到，执行位左移指令和位右移指令的梯级条件是选择了计数器的 DN 位在前面，计时器的 DN 位在后面，将位变化频度较高的放在后面，位变化频度较低的放在前面，这是程序扫描优化所要考虑的，虽然我们在位指令编程的章节已经讨论过，但请留心这些细节。

位移动指令要求参加移动的数据对象必须是双整字数组，鉴于此例，我们选了一个长度为 1 的双整字数组，尽管我们需要用的不足一个双整字，只有 4 个位参加移动，并且只能是从 0 位开始的这个双整字的低 4 位移动。

位左移指令和位右移指令只能对数组的字进行移动，最后的输出需要送往输出模块，只能转换一下。简单的用 MOV 指令是有风险的，除非操作对象正好是整个模块的输出点，因为只涉及输出的 4 个点。采用位传送的 BTD 指令指定传送部分和传送位置，也可以采用屏蔽传送指令 MVM 选择部分位传送，如图 10-6 所示。

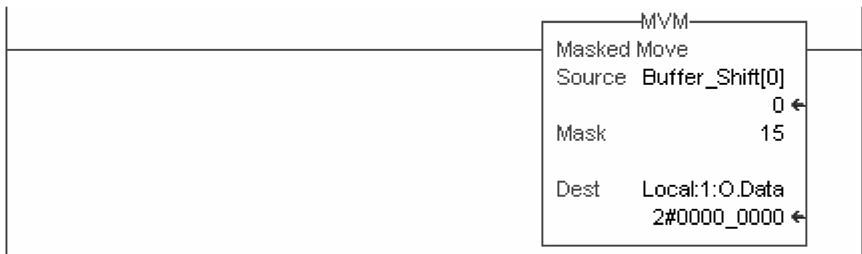


图 10-6　采用屏蔽传送指令 MVM 选择部分位传送

程序运行之初，需要预先设定最低位为 1，可使得 DO/0 的灯点亮。想要在移动数组的最低位设置为 1，可不可以用如图 10-7 所示的位操作的梯级来实现呢？



图 10-7　锁定最低位

在线反复进行几次调试就能看出问题，仅仅对 Buffer_Shift［0］双整字的最低位置 1，的确能提供一个初始的灯亮状态，但却不能清除上次移动的残留状态，我们不能确定前次正好停留在哪个位。MOV 指令或 MVM 指令可以解决这个问题，对一个字传送一个 1 意味着除了最低位置为 1，其余位统统清除为零。当然，目前我们的这个操作只有一个字，如果是数组，要求的初始值不是最低位为 1，那就要另谋出路了，初始化的操作将复杂一些。

此外，移位指令中所分配的控制结构数据的指针项没有意义，甚至在指令参数中都不予体现，这是惟一一个数组操作控制结构无须指针的特例，如果有兴趣，在数据库的标签中可以观察到，当移位指令使能时指针等于长度，移位指令未使能时指针等于 0。指针的状态一般来说没有实用价值。

位移指令常常被用作记住流水线操作对象的存在标志，当一个流水线上的操作对象到达感应位置时，感应开关接通，该位置 1，表示操作对象的存在，操作对象流动至操作区域则做相应操作；当没有操作对象到达，感应开关在置 0 状态，表示操作对象不存在，无操作对象的操作区域不做任何动作。

如图 10-8 所示编写的梯级逻辑就是一个这样的实例，步进脉冲定时探测是否有瓶子经

过，当有瓶子经过时，传感器接通，装载位置 1，随着 BSL 指令的不断执行，该瓶子携带的标志位移动，移动到 5 的位置，进入装填例程执行；移动到 11 的位置，进入封装例程执行；移动到 27 的位置，进入下线例程执行。当探测到没有瓶子经过，装载位为 0，瓶子的标志位 0 移动，所有的位置均无操作，没有满足梯级条件的例程调用。

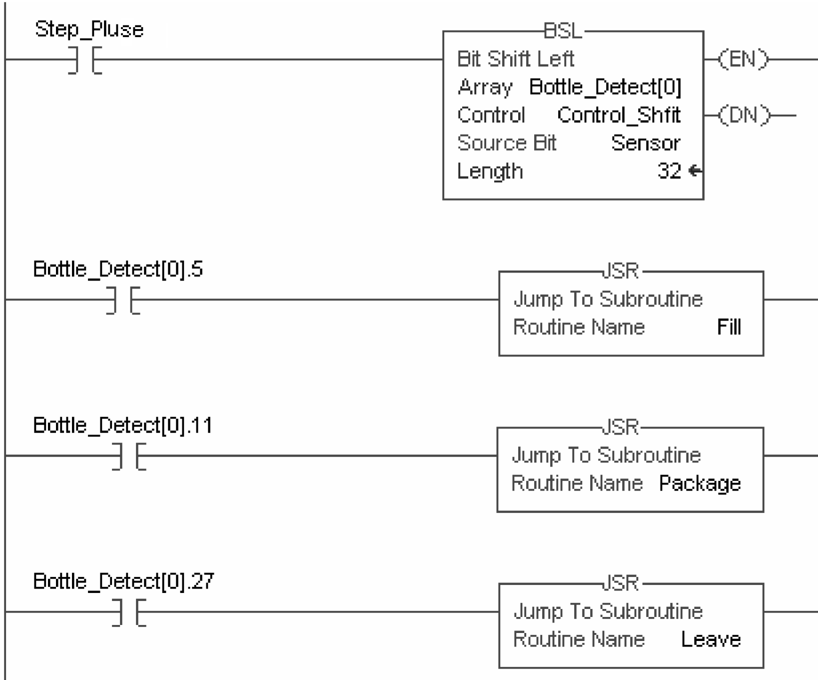


图 10-8　移位操作实例

这里 5 的位置、11 的位置和 27 的位置是由瓶子移动的距离决定的，瓶子移动的速度恒定，乘以探测时间间隔便得到了距离，这些数据可以通过现场调试而获得。

移位指令的编程大多出现在早期的程序，深受单板机编程思想的影响，目的在于节省内存和加快速度，并减轻 CPU 的负担。在现今内存充裕和 CPU 运行速度快的控制器系统中，无须过于考虑节省，而是更兼顾对于控制过程表达的准确和直接，以及程序的严谨，采用移位指令解决问题越来越少了，日益丰富的指令系统提供了更方便的处理方式。

10.2　堆栈指令先入先出 FFL/FFU 和先入后出 LFL/LFU 的编程

被称为堆栈指令执行的寄存器批量数据操作是对寄存器区域数据存放和数据进出的管理，这是对寄存器数据块的操作，由一个负责数据装载的指令 FFL 和一个负责数据卸载的指令 FFU 相互配合完成。这两条指令必须成对使用，因装载和卸载的操作关联，它们必须共同使用一个数组操作的控制结构数据。所谓操作关联，指的是操作对象是同一个数组的数据；确定数组位置的指针会跟随装载或卸载的操作动作而浮动。

根据堆栈数组先入先出的原则，装载指令 FFL 在梯级条件跳变时，控制结构数据的 POS 加 1，装载数据到指针 POS 所指向的数组元素；卸载指令 FFU 在梯级条件跳变时，卸载最前

面的数据，同时全体数据前移，控制结构数据的 POS 减 1。这个控制结构数据的指针变化非常像可逆的增减计数器的动作，你应该还没有忘记，前面我们提过，数组操作所需的控制结构数据是从计数器演变过来的，或者说是一个特殊的专用计数器。

控制结构数据状态位也在堆栈指令中具有特殊的含义，完成位置位，其含义是数组堆栈满，不能继续对数组装载，此时的装载操作执行无效；栈空位置位，数组不能继续卸载，返回 0 给目标操作数。如果对数组堆栈操作有限定要求，不妨把这两个位用作限制条件，或通过这些位状态来满足某种特定的需求。

梯级逻辑编写如图 10-9 所示，本条装载指令 FFL 的执行是当梯级条件 FIFO_Load 跳变，控制结构数据的指针 Control_LU. POS 加 1，将源操作数 Source_FIFO 的数据装载至指针所指的堆栈数组元素 Array_FIFO［6］，完成数据装载操作。

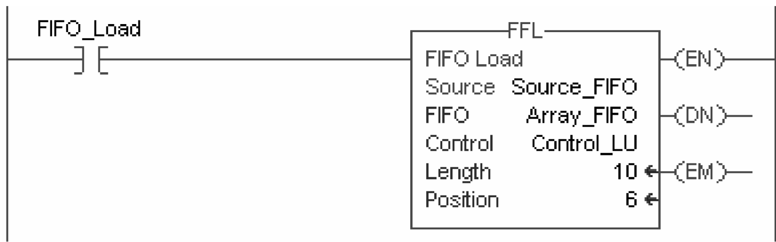


图 10-9　堆栈装载的梯级逻辑

梯级逻辑编写如图 10-10 所示，本条卸载指令 FFU 的执行是当梯级条件 FIFO_Unload 跳变，执行卸载数据操作，将堆栈数组最前面的数据弹出至目标操作数 Dest_FIFO，并且数组全体数据前移一个数据单元，同时 Control_FIFO. POS 减 1，将 0 充填原来指针所指的数据单元，即刚移走的数据单元 Array_FIFO［6］。

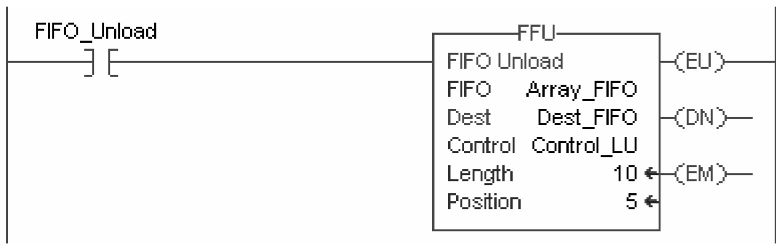


图 10-10　堆栈卸载的梯级逻辑

堆栈数据装载和卸载操作的示意图如图 10-11 所示。定义 10 个双整字的数组作为数据堆栈区域，装载指令 FFL 操作将源操作数送至指针所指的数组数据单元；卸载指令 FFU 操作将数据送至目标操作数，然后全体数据前移，置 0 充填原指针所指的数据单元，即堆栈数据的最后一个数据单元。

通过装载和卸载两条堆栈指令的配合使用，可以完成先入先出的数据堆栈存取，因而常常用于更新式的数据采集。其做法是，推掉一个最旧的数据，采集一个新数据补充，总是保存指定长度的最后的一组数据。

实际运用中常常会遇到这样的需求，针对一个对象按照一定的时间间隔采集数据，把这个数据存放在指定的数据缓冲区，并保持最后的新鲜数据，每当采集到一个新的数据，就推

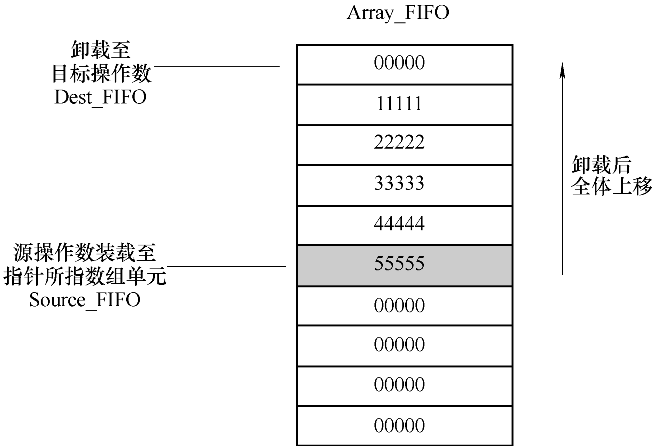


图 10-11　堆栈数据装载和卸载操作的示意图

掉最前面的旧数据。这是典型的数据堆栈的管理，用先入先出的堆栈指令 FFL 和 FFU 来完成实在是太合适不过了。

假定有一个这样的需求实例，每隔 2s 从模拟量输入通道 0 采集一次数据，将数据存放在指定的堆栈数组，然后将这个堆栈数组的数据送到人机界面列表显示，当这个堆栈数组最初是空的时候，一直执行装载数据的动作，不卸载数据，直到堆栈数组装满，然后总是推掉最前面的数据，将刚采集的数据充填在最后一个数据单元，一直保持最后的 20 个数据。编写的梯级逻辑如图 10-12 所示。

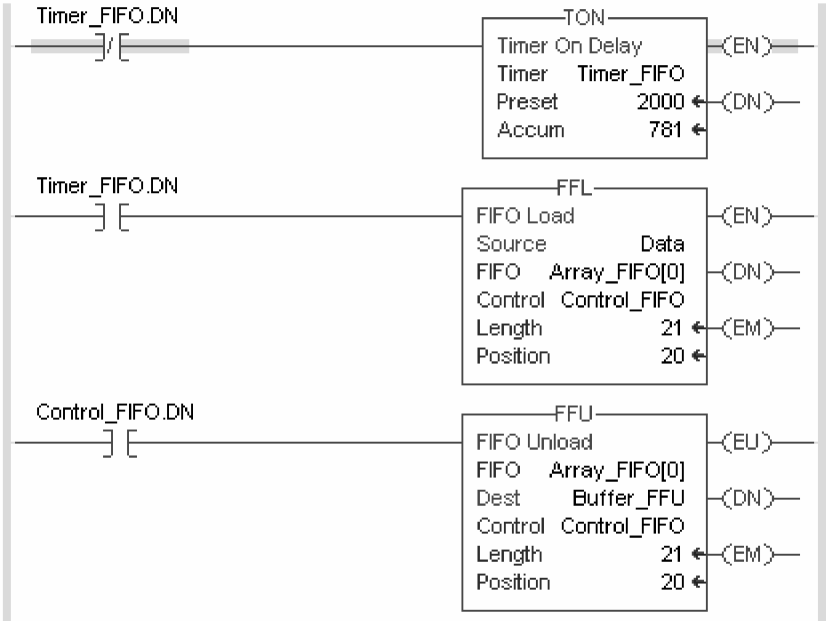


图 10-12　更新式数据采集的梯级逻辑

编写一个自复位的计时器 Timer_FIFO，其完成位 DN 提供每 2s 一次的工作脉冲，这个工作脉冲为堆栈数组的装载指令提供所需的梯级条件跳变。

堆栈数组的指针最初为零，当计时器产生第一个脉冲，装载 FFL 指令的梯级条件跳变，Control_FIFO. POS 加 1，指针 POS 指向 1，从数据源采集来的数据装入第一个数据单元。如此类推，每当梯级条件跳变一次，指针 POS 加 1，采集的数据装入指针所指的数组数据单元，直到栈满，也即 Control_FIFO. DN 位置位。栈满位作为 FFU 的梯级条件，根据先入先出的原则，FFU 卸载最前面的一个数据，并将所有的数据往前移动，将 0 置入腾出的最后一个数据单元，同时 Control_FIFO. POS 减 1。这种控制结构数据标签的 POS 增减的修改情形就是前面提到的相似于可逆增减计数器的控制结构数据指针。

正如前面提及的，装载指令 FFL 和卸载指令 FFU 共有同一个控制结构数据 Control_FIFO，成对操作的指令 FFL 和 FFU 不但操作对象相同，而且操作动作关联。它们对同一个堆栈数组操作，修改同一个控制结构数据标签的指针，这些恰恰是控制结构数据的参数所示，堆栈数组的长度和指针。

我们把这个程序下载到控制器去运行测试，观察它的堆栈数组，你无论如何也不会相信自己的眼睛，怎么看数据都是少一个。也许堆栈数组的卸载操作不该编写在堆栈数组的装入操作的梯级后面，应该让它先卸出，再装入。当把卸出指令的操作梯级改为放置在装载指令操作的梯级前面，重新测试一下，情况仍然没有改变。

其实这个最后的数据是存在的，不过存在的时间极短，如果将装载指令 FFL 编写在前面梯级，后面梯级的卸载指令 FFU 紧跟着就操作卸载并移动了；如果将装载指令 FFL 编写在后面梯级，将延迟一个扫描周期的时间，前面梯级的卸载指令 FFU 才操作卸载并移动。但对于我们观察数据来讲是一样的，这就是为什么刚才我们调换两条指令执行的顺序结果仍然没有改变的原因。

我们总是无法看到瞬间存在的最后一个数据，所看到的是卸载指令 FFU 卸载操作之后的数组最后一个数据元素填补的0，这个数据每持续2s 将闪变一次，但我们感觉不到，这样给了我们总是少一个数的错觉。主要原因在于我们用 FFL 装载后的完成位作为卸载 FFU 的梯级条件，虽然这样的梯级关系简单明了，执行目的一看就知，但卸载指令 FFU 补 0 的操作让我们不能看到最后一个数据。如果只是为了得到你想要的数据长度 N，大可不必用复杂的梯级条件来解决这个问题，直接将数组操作长度设为 N+1 就可以，注意创建数组的长度也必须是 N+1 或更大，否则例程执行将报错而引起停机。

堆栈数组先入先出的操作，先这么做着吧，直到有一天你读了别人写的程序，也是 N+1 的数组操作长度，遂发出会心的笑，原来那位台兄也是屡试不爽，委曲求全那，好在这个数组长度也就它自己消化，无伤大雅，我们就接受了吧。这也算是编程中常见的一种迁就，但我们必须确定，这个长度设置不会影响到其他。

可以想象一下，卸载指令 FFU 执行的最前面的一个数据卸载之后，全体向前移动一个单元，这种操作不就是前面谈到的 COP 指令的技巧性运用么，料想当初没有堆栈指令 FFL 和 FFU 时，就是用 COP 指令来解决问题的，或许组成堆栈指令 FFU 的宏中应该是包括了这个 COP 动作吧。这里，我们再一次看到指令功能代替了技巧的运用。

我们再来看看，如果对堆栈数组的目标操作数和源操作数进行特殊的选定，就可以实现数组的数据循环了。为简单起见，假定有一个需求是希望得到一组 5 个双整字数据的循环移动。模仿刚才的做法，固然可以得到数据的循环，既然不存在数组堆栈空和堆栈满的不同动作，我们就避免使用控制结构数据的 DN 位，因为它总是带来一些困惑，令读例程的人们难

以理解，这个需求是一开始就处在堆栈完成状态，编写的梯级逻辑如图 10-13 所示。

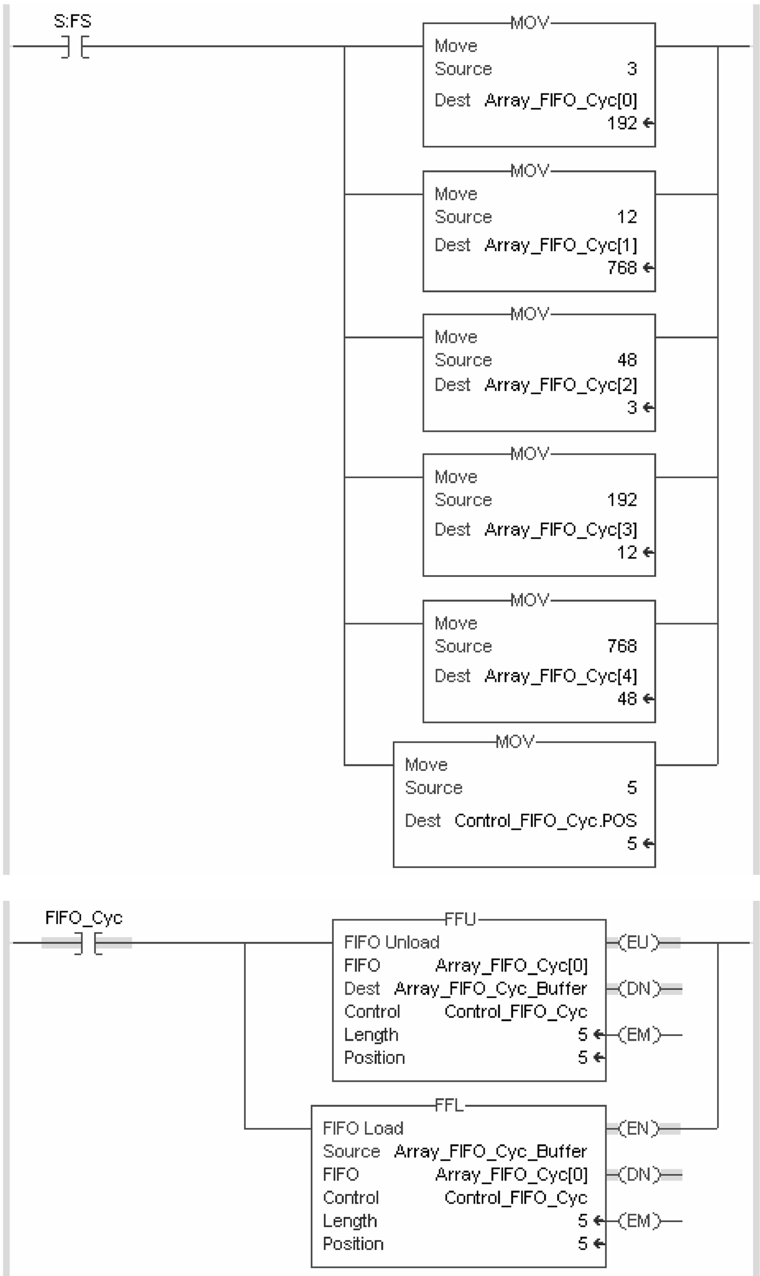


图 10-13　循环数据设置和执行梯级逻辑

程序扫描的初始化先将数组的数据预装好（这部分梯级逻辑也可以编写在初始化例程中），这个数据的初始状态可能是预定的数据值，也可能是某状态量字演变过来的十进制数字表达，通过 MOV 指令的立即数来进行例程执行的数据设置，为系统的可靠性提供保证。

同时，将堆栈指令 FFL 和 FFU 共用的指针 Control_FIFO_Cyc. POS 预设为 5，这确保了一开始就能进入循环并保证了数据的首尾相接，数据循环的指针永远定格在长度 5 上。

以循环步进动作 FIFO_Cyc 的位状态作为堆栈指令 FFU 和 FFL 执行的梯级条件，当梯级条件跳变，首先执行的是卸载指令 FFU，将 Array_FIFO_Cyc［0］卸出，送到循环缓冲数据单元 Array_FIFO_Cyc_Buffer，同时 POS 减 1，全体数据前移，并补 0 给最后一个数据单元，即 Array_FIFO_Cyc［4］；紧跟着执行的装载指令 FFL 立刻将 POS 加 1，此时指针指向 5，循环缓冲数据 Array_FIFO_Cyc_Buffer 作为装载指令 FFL 的源数据被送到指针所指的数据单元，即刚被补 0 的 Array_FIFO_Cyc［4］。

这个过程精密而准确，每一个步骤都清楚地印证了装载指令 FFL 和卸载指令 FFU 的执行动作。试着将装载指令 FFL 和卸载指令 FFU 交换一下位置，即执行顺序的改变，得到的结果便不尽人意了，因为这样执行的动作进程不是我们所要的循环。

这样的卸载指令 FFU 和装载指令 FFL 的交替操作，实现了将数组的最前面的数据单元传递到最后一个数据单元，如此循环，如果将数组 Array_FIFO_Cyc 在别的地方引用，这将是一个循环的数组，是控制过程的动作所需要的。

还有一套先入后出堆栈操作指令 LFL 和 LFU，其管理原则是按指针装载数据和按指针卸载数据，装载指令 LFL 的操作跟 FFL 是完全一样的，不同的是卸载的时候，按照指针所指卸出，即后入先出。先入先出和先入后出两对指令混合着用也是可以的。

现在我们试试能否将一个数组的最后一个单元传递到另外一个数组的最前面的一个单元，即将原有的数据颠倒顺序。第 9 章中我们曾使用数组排序指令 SRT 对数组 Array_SRT 进行排序，这是一个升序的执行结果，如果我们想要一个降序的结果，能否将排序的结果颠倒顺序呢？编写的梯级逻辑如图 10-14 所示。

梯级条件 Start_SRT 转为真，为倒序梯级逻辑提供了条件，两条 MOV 指令为 LFU 指令和 LFL 指令的指针设置了初始化数据，卸载指令给予堆栈满的状态，装载指令给予堆栈空的状态，卸载指令 LFU 从最后面一个数据开始卸载，装载指令 LFL 从最前面一个数据开始装载。

这两条指令是分别独立操作的，既不是相同的操作对象，操作上也没有指针的关联，这里不是成对使用的例子，故各自使用自己独立的结构数据标签 Control_SRT1 和 Control_SRT2。

保持型的堆栈指令需要使能位跳变才能执行，这里采用解锁指令复位使能位的方法，迅速简单地连续操作，所以 LFU 和 LFL 指令都是无梯级条件的。

装载指令 LFL 的完成位则作为继续循环或离开循环的判断条件。离开循环时复位启动倒序操作的 Start_SRT 标签和装载指令控制结构数据标签，以准备下一次的启动，复位控制结构数据标签也是为了消除复位条件，以便 Start_SRT 能再次启动。Start_SRT 的置位状态只会存在一个扫描周期。

倒序前后的数据表排列如图 10-15 所示。

对照数据表可以看出数据堆栈的搬运结果，这和通常的 MOV 指令和 COP 指令执行结果不一样，搬走后的数据，留下的是 0，源数据不复存在。仅仅是将数组倒序作为操作目的，应该在每次倒序之前将操作对象复制到源数据。

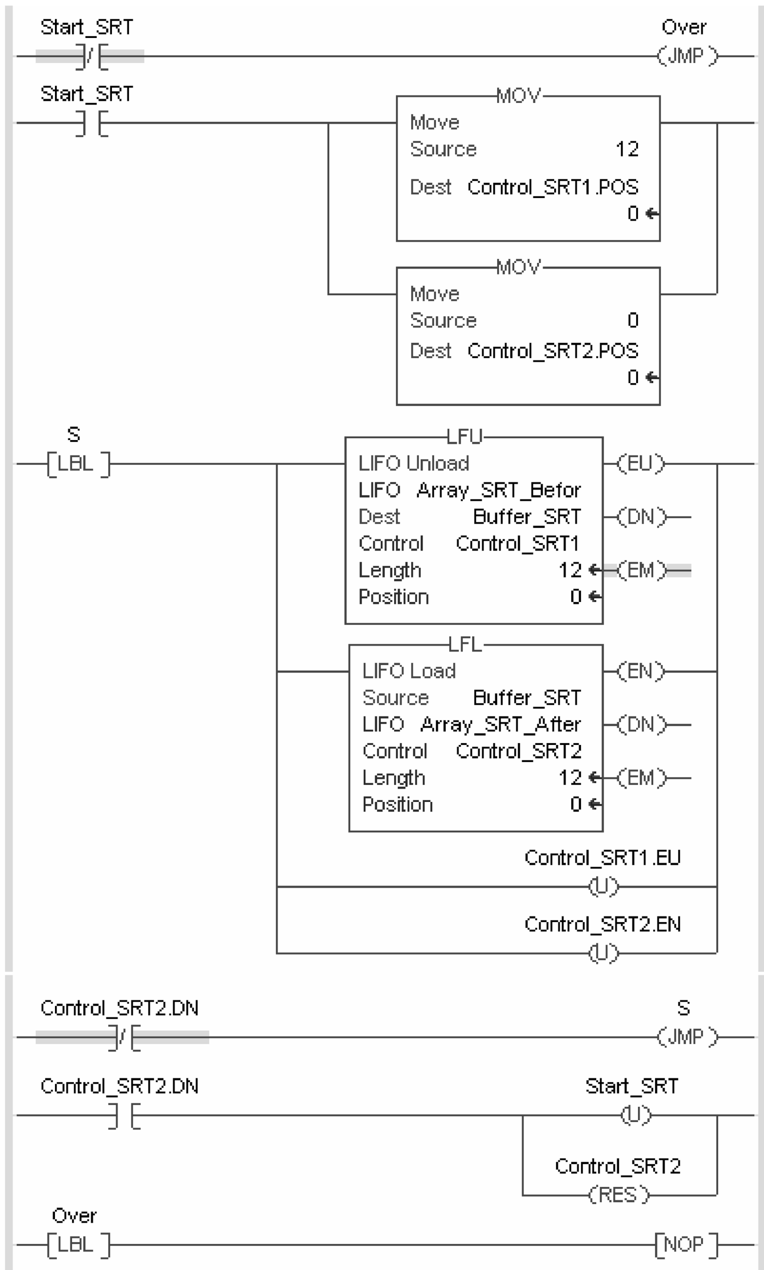


图 10-14　颠倒数组顺序的梯级逻辑

倒序前

Array_SRT_Befor	{...}
Array_SRT_Befor[0]	25
Array_SRT_Befor[1]	89
Array_SRT_Befor[2]	123
Array_SRT_Befor[3]	166
Array_SRT_Befor[4]	182
Array_SRT_Befor[5]	187
Array_SRT_Befor[6]	234
Array_SRT_Befor[7]	247
Array_SRT_Befor[8]	267
Array_SRT_Befor[9]	289
Array_SRT_Befor[10]	398
Array_SRT_Befor[11]	456

Array_SRT_After	{...}
Array_SRT_After[0]	0
Array_SRT_After[1]	0
Array_SRT_After[2]	0
Array_SRT_After[3]	0
Array_SRT_After[4]	0
Array_SRT_After[5]	0
Array_SRT_After[6]	0
Array_SRT_After[7]	0
Array_SRT_After[8]	0
Array_SRT_After[9]	0
Array_SRT_After[10]	0
Array_SRT_After[11]	0

倒序后

Array_SRT_Befor	{...}
Array_SRT_Befor[0]	0
Array_SRT_Befor[1]	0
Array_SRT_Befor[2]	0
Array_SRT_Befor[3]	0
Array_SRT_Befor[4]	0
Array_SRT_Befor[5]	0
Array_SRT_Befor[6]	0
Array_SRT_Befor[7]	0
Array_SRT_Befor[8]	0
Array_SRT_Befor[9]	0
Array_SRT_Befor[10]	0
Array_SRT_Befor[11]	0

Array_SRT_After	{...}
Array_SRT_After[0]	456
Array_SRT_After[1]	398
Array_SRT_After[2]	289
Array_SRT_After[3]	267
Array_SRT_After[4]	247
Array_SRT_After[5]	234
Array_SRT_After[6]	187
Array_SRT_After[7]	182
Array_SRT_After[8]	166
Array_SRT_After[9]	123
Array_SRT_After[10]	89
Array_SRT_After[11]	25

图 10-15　倒序前后的数据表排列

10.3　顺序器输出指令 SQO、顺序器输入指令 SQI 和顺序器装载指令 SQL 的编程

顺序器指令可以称得上是最早的数据块处理指令了，面对工艺过程控制的大量逻辑步进关系，当初内存又是如此的金贵，在不得不竭尽全力地节省内存的同时，还要兼顾大量的逻辑处理，顺序器指令这种来自于硬件的启发就这样产生了。自从有了 PLC 的那天就有了如此复杂的指令，但是它的确管用。

顺序器输出指令将按步骤输出的状态预先存放在被称为顺序器输出数组的数据块中，当步进进程变化，即指令的梯级条件跳变，顺序器输出指令执行，将指针指向的数据单元传送给输出目标操作数，这个目标操作数控制着外部设备，如此周而复始地重复。不难想象，如果这些逻辑输出分散地用梯级逻辑表现，不但消耗的内存会大量增加，并且不能集中看到设备的步进过程输出状态的关系，这些步进过程状态在生产工艺过程的列表中一目了然的形式在顺序器输出数据表中再次体现。

让我们来看一个这样的应用实例，有一个包装机，其包装过程可以分为 7 个生产步骤，这 7 个生产步骤都会对一个输出模块的输出点产生不同的输出状态去控制外部的设备运转状态，每前进一个步骤而产生的输出各点的状态，其过程没有规律性的变化。受限于编程实验设备的 I/O 点数，我们先假定这个离散量输出模块有 4 个输出点，这是我们要进行输出处理的对象。每个步骤的 4 位状态见表 10-1。

表 10-1　每个步骤的 4 位状态

	第 3 位	第 2 位	第 1 位	第 0 位
第一步骤	1	0	0	1
第二步骤	0	1	1	0
第三步骤	1	0	0	0
第四步骤	0	1	0	0
第五步骤	1	0	0	1
第六步骤	1	1	0	1
第七步骤	1	1	1	1

创建一个顺序器输出数组标签，7 个长度的双整字数组。我们把这个在控制过程中将出现的 7 个控制状态预存到顺序器输出数组中，如图 10-16 所示。

⊟Array_SQO	{...}
⊞Array_SQO[0]	2#0000_0000_0000_0000_0000_0000_0000_0000
⊞Array_SQO[1]	2#0000_0000_0000_0000_0000_0000_0000_1001
⊞Array_SQO[2]	2#0000_0000_0000_0000_0000_0000_0000_0110
⊞Array_SQO[3]	2#0000_0000_0000_0000_0000_0000_0000_1000
⊞Array_SQO[4]	2#0000_0000_0000_0000_0000_0000_0000_0100
⊞Array_SQO[5]	2#0000_0000_0000_0000_0000_0000_0000_1001
⊞Array_SQO[6]	2#0000_0000_0000_0000_0000_0000_0000_1101
⊞Array_SQO[7]	2#0000_0000_0000_0000_0000_0000_0000_1111

图 10-16　顺序器输出数组的数据设置

下面编写一个顺序器输出指令的梯级，如图 10-17 所示。这个顺序器输出指令使用图 10-16 数据设定的输出数组 Array_SQO，通过只让低 4 位传递的屏蔽字，在梯级条件的控制下逐步将 4 个位状态送到同一个输出模块，形成 7 个步骤的顺序控制。

给予这个梯级一个梯级条件，诸如这样的指令是需要梯级条件的跳变才能够执行的，每当生产过程前进一步，步进梯级条件就跳变一次，跳变触发令顺序器输出指令将指针 POS 加 1，并将指针所指单元的双整字输出，传送到输出模块，给相应的输出点提供当前步的控制状态。

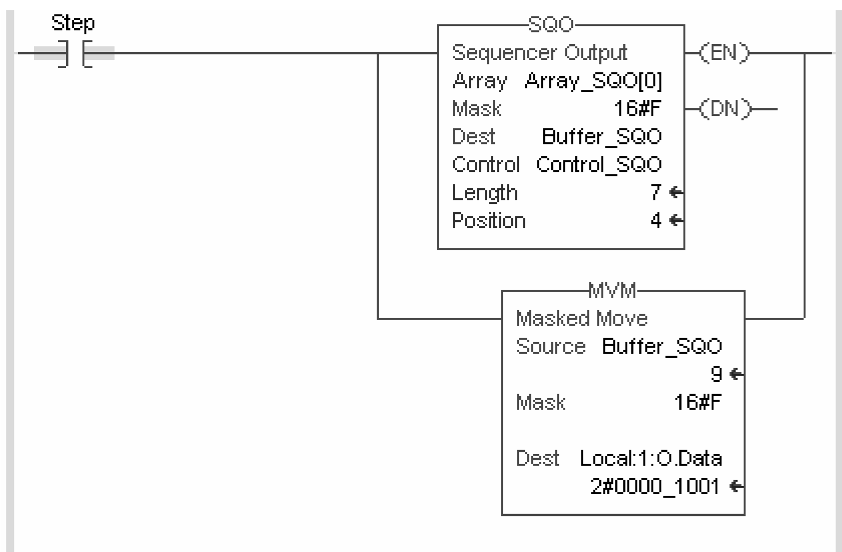


图 10-17　顺序器输出指令的梯级

这里输出目标字在 SQO 指令中，要求双整字（查看在线帮助或指令集手册）。此例中，离散量输出模块的输出数据字是 8 位，只好建立一个缓冲字 Buffer_SQO，然后通过传送指令传递数据。如果是 32 位的输出字的输出模块，指令输出目标字直接引用模块地址。

使用带屏蔽 MVM 指令传送到输出模块，而不是使用 MOV 指令，这样更为可靠，且不会影响到输出模块的其他位，尽管我们这里正好是 4 位输出，但要养成好的习惯，决不妨碍其他的位状态。

MVM 指令可以单独编写一个梯级跟在后面，但这样每次扫描都要执行，如果并在 SQO 指令后面，则只需在梯级条件成立期间传送。如果梯级条件持久而不是突跳，希望减少执行，则可加上 ONS 指令限制，这又是一个可关注的细节。

改变梯级条件，观察顺序器输出 SQO 的执行过程，每当梯级条件从 0 跳到 1，前沿触发令顺序器输出指令的指针加 1，相应单元数据被送到目标字，当 POS 为 7 时，梯级跳变，指针回到 1，重新开始新的一轮顺序输出。

那么，进入下一生产步骤的梯级条件是如何确定的呢？显然要根据现场的反馈，才能决定是否进入下一个步骤，这些现场的反馈来自于传感器输入信号，即输入模块的信息。最精彩的莫过于顺序器输出指令 SQO 与顺序器输入指令 SQI 的配合使用了。

SQI 指令是少见的一条输入指令，位于梯级的左侧，每次扫描指令都被检测，决定梯级条件是否成立。所谓检测，是预先存放的参考数组按指定的顺序与检测对象进行比较，当比较结果相同，梯级条件成立；比较结果不同，梯级条件不成立。这个检测对象显而易见就是来自现场传感设备采集的输入模块状态。

继续前面包装机的实例，为其 7 个步骤进程的顺序输出，配合使用 SQI 指令来提供步骤进程的梯级条件。

首先，我们创建一个状态参考数组，受限于编程实验设备 I/O 点数，我们先假定采集步骤完成状态的离散量输入模块有 6 个输入点的状态是我们要进行比较的对象。每个步骤完成

的 6 位输入状态见表 10-2。

表 10-2　每个步骤完成的 6 位输入状态

	第 5 位	第 4 位	第 3 位	第 2 位	第 1 位	第 0 位
第一步骤	0	0	1	1	0	1
第二步骤	0	1	0	1	1	0
第三步骤	1	0	1	1	0	1
第四步骤	1	0	0	0	1	1
第五步骤	0	0	0	1	0	0
第六步骤	1	0	1	0	1	1
第七步骤	1	1	0	1	0	1

创建一个顺序器输入参考数组标签，7 个长度的双整数组。我们把这个在控制过程中 7 个步骤完成所探测的输入状态预存到这个数组中，以供对比，如图 10-18 所示。

− Array_SQI	{...}
+ Array_SQI[0]	2#0000_0000_0000_0000_0000_0000_0000_0000
+ Array_SQI[1]	2#0000_0000_0000_0000_0000_0000_0000_1101
+ Array_SQI[2]	2#0000_0000_0000_0000_0000_0000_0001_0110
+ Array_SQI[3]	2#0000_0000_0000_0000_0000_0000_0010_1101
+ Array_SQI[4]	2#0000_0000_0000_0000_0000_0000_0010_0011
+ Array_SQI[5]	2#0000_0000_0000_0000_0000_0000_0000_0100
+ Array_SQI[6]	2#0000_0000_0000_0000_0000_0000_0010_1011
+ Array_SQI[7]	2#0000_0000_0000_0000_0000_0000_0011_0101

图 10-18　顺序器输入参考数组的数据设置

顺序器输入指令 SQI 的工作是监视当前步状态的完成。每当梯级被扫描时，SQI 指令中的输入参考数组 Array_SQI 与 Buffer_SQI（来自离散量输入模块的对现场的状态检测）进行比较，这些被检测的状态位在不同的扫描周期陆续地达到了当前步的状态，总会有某次扫描周期，最后一个检测位也达到了应有的状态，比较的结果判定相等，梯级条件成立。获得梯级条件触发的顺序器输出指令，POS 加 1，送出下一步输出控制状态，生产过程进入下一步的执行。与此同时，顺序器输入指令开始监视下一步，并等待检测结果相等的时机出现。

现在要解决的问题是，顺序器输入指令如何进入到下一步。我们已经知道，所有的数组指令的动作都是控制结构数据标签的指针来决定执行位置的，你可能已经注意到图 10-19 中的梯级逻辑编程了，这个 SQI 指令竟然跟 SQO 指令共享一个控制结构数据标签。这正是这两条指令配合使用的绝妙之处！SQI 是一条输入指令，虽然每次梯级扫描都被检测，它永远也不可能获得一个触发去修改自己的指针 POS。由于 SQI 是不能内部修改的，只能依靠外部修改，这自然需要额外编写梯级逻辑套用同步长的 SQO 指令的指针 POS，真是得来全不费功夫。

同样的道理，顺序器输入指令也是只能使用 32 位的双整字，我们的离散量输入模块的输入数据字是 8 位的，只好建立一个缓冲字 Buffer_SQI，然后通过带屏蔽的 MVM 指令传递。注意，这个传递必须放置在 SQI 指令所在梯级之前，并且每次扫描周期都要传送，以确保输入模块的信息及时映射到缓冲字 Buffer_SQI。尽管 SQI 的屏蔽字作用确保了参与比较的输入

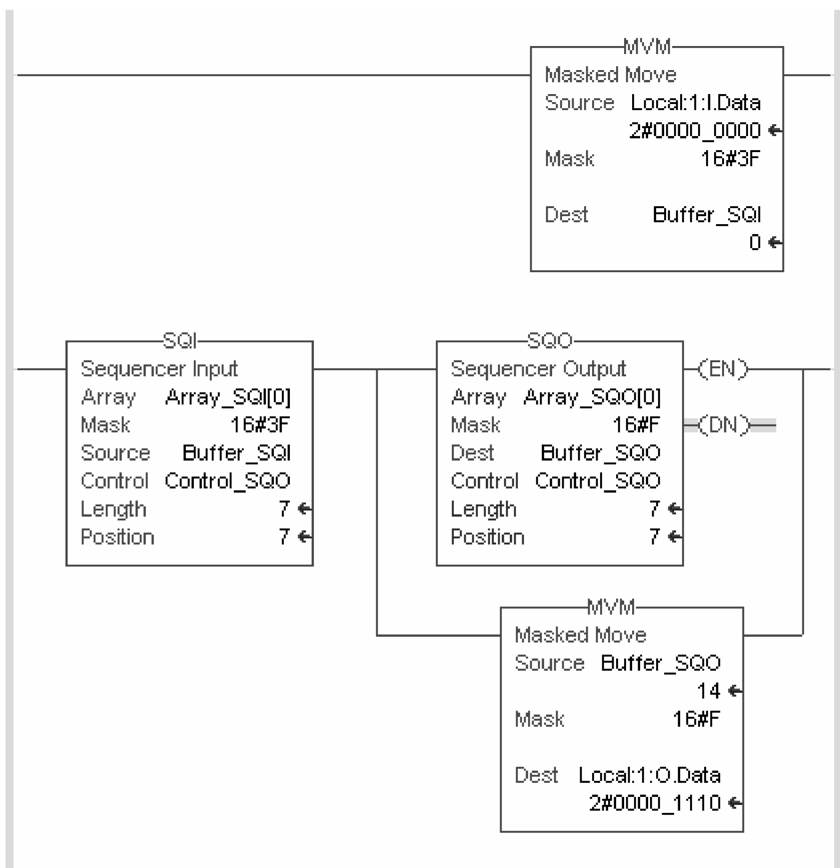


图 10-19　顺序器输出与顺序器输入的配合运用

字只会在低 6 位进行比较，还是使用了 MVM 指令防止干扰。

一对 SQI 和 SQO 指令只对一个输出模块的状态执行，对生产工艺过程的控制远远不止十几个点的顺序控制，这时你可以看到很多对 SQI 和 SQO 指令在同时控制着整个系统。

如果顺序器输出的数组是一个根据生产工艺变化的数据，而这个数据通常由操作员修改，这时顺序器装载指令 SQL 就派上了用场。操作员在人机界面上键入的数组参数为 SQL 的源，当界面 Entry 操作时，作为 SQL 的梯级条件令顺序器装载动作执行，从而逐个地修改了输出数组文件，如图 10-20 所示。顺序器装载指令 SQL 的工作与顺序器输出指令 SQO 很相似，每当梯级条件跳变，POS 加 1，源数据送往指针所指的数据单元。Buffer_SRL 标签是人机界面上操作员键入数据所访问的标签，Entry 则是确认数据键入的 BOOL 量所访问的标签。

如果顺序器输出的数组是固定不变的，意味着这些数据在编程的时候事先设定了，为了系统的可靠性，有经验的工程师往往会在初始化的例程里编写一段梯级逻辑，专门用来设置数据。这样，万一数据设置不小心被改动或错乱，只须重新启动运行就可以恢复正常的设置。

通常这种设定用 MOV 指令完成，把要设置的数作为立即数传送，请注意这些立即数都

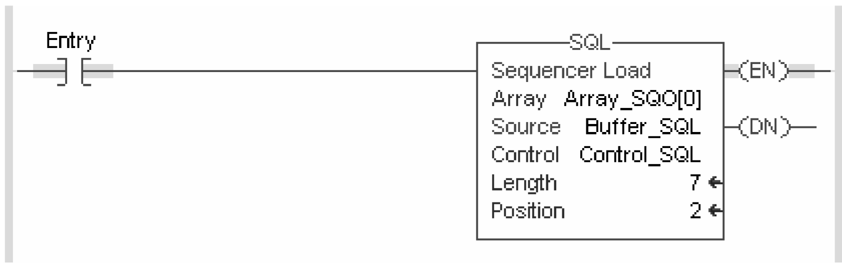


图 10-20 顺序器装载梯级逻辑

是十进制数。把刚才状态量的二进制所表示的数据形式转为十进制数的形式，如图 10-21 和图 10-22 所示。

Array_SQO	{...}	Decimal
Array_SQO[0]	0	Decimal
Array_SQO[1]	9	Decimal
Array_SQO[2]	6	Decimal
Array_SQO[3]	8	Decimal
Array_SQO[4]	4	Decimal
Array_SQO[5]	9	Decimal
Array_SQO[6]	13	Decimal
Array_SQO[7]	15	Decimal

图 10-21 顺序器输出状态量设置的十进制数

Array_SQI	{...}	Decimal
Array_SQI[0]	53	Decimal
Array_SQI[1]	13	Decimal
Array_SQI[2]	22	Decimal
Array_SQI[3]	45	Decimal
Array_SQI[4]	35	Decimal
Array_SQI[5]	4	Decimal
Array_SQI[6]	43	Decimal
Array_SQI[7]	53	Decimal

图 10-22 顺序器输入状态量设置的十进制数

通过数据表的换算，现在应该知道 MOV 指令传送什么样的立即数据了，编写梯级逻辑如图 10-23、图 10-24 所示。利用数据库中的数据来获得换算的关系，避免了人工计算可能出现的错误，因为这是控制器自己对数据的识别。

这是针对顺序器指令设置数据的一个例子，大凡遇到系统的数据预先设置，建议你不要直接在数据标签中设置数据，而是在初始化程序里执行 MOV 指令这样不但可靠，也能让读

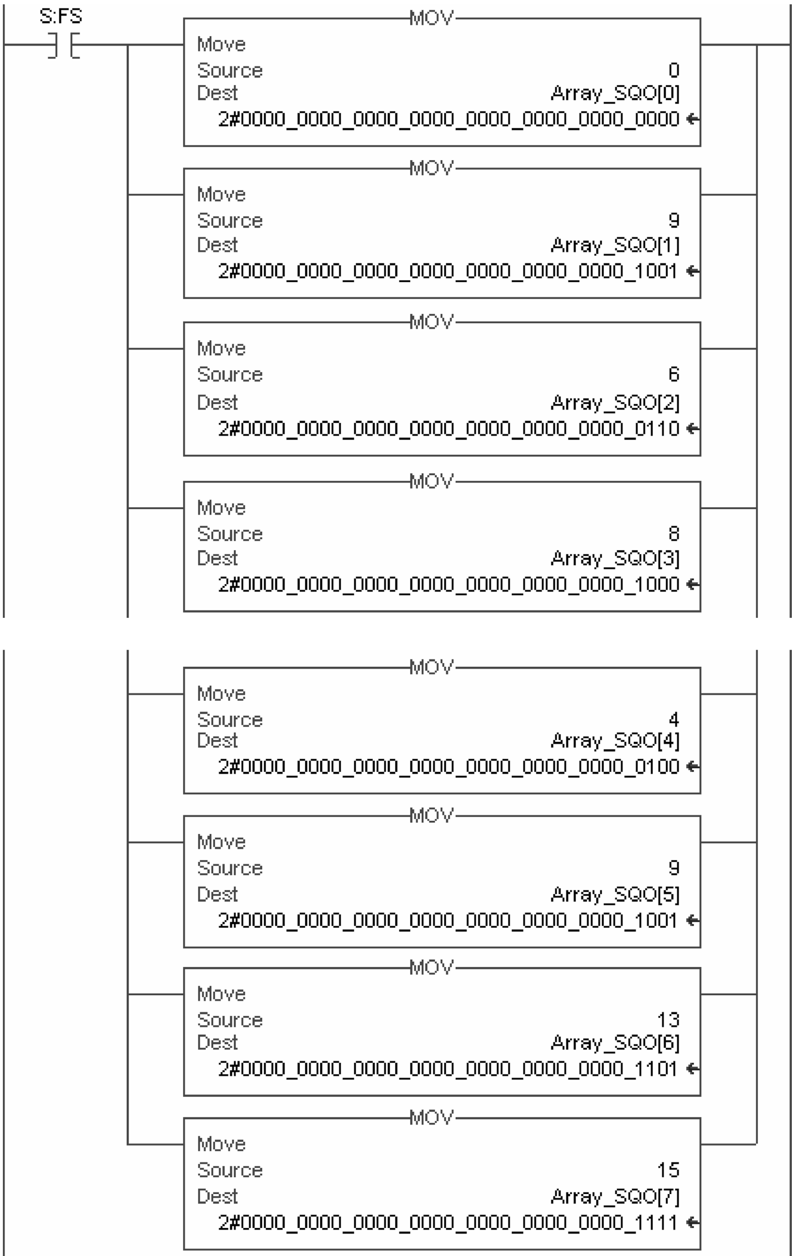


图 10-23　传送顺序器输出数组的设定值

程序的人能够了解那些数据为什么预先设置，在梯级执行的注释中可加以说明，并能看到设置的数据值。有位资深项目工程师告诉我，他在项目中规定初始化程序的编写一律要求采用语句编程，因为只有语句编程才不会丢失注释信息，可见初始化信息的解释是多么重要。

关于顺序器输出指令 SQO 和顺序器装载指令 SQL 还有一个很麻烦的问题，就是 0 步的输出，所以指令在初次进入顺序循环时，要很谨慎地确定在 0 步还是在 1 步。通常我们在数

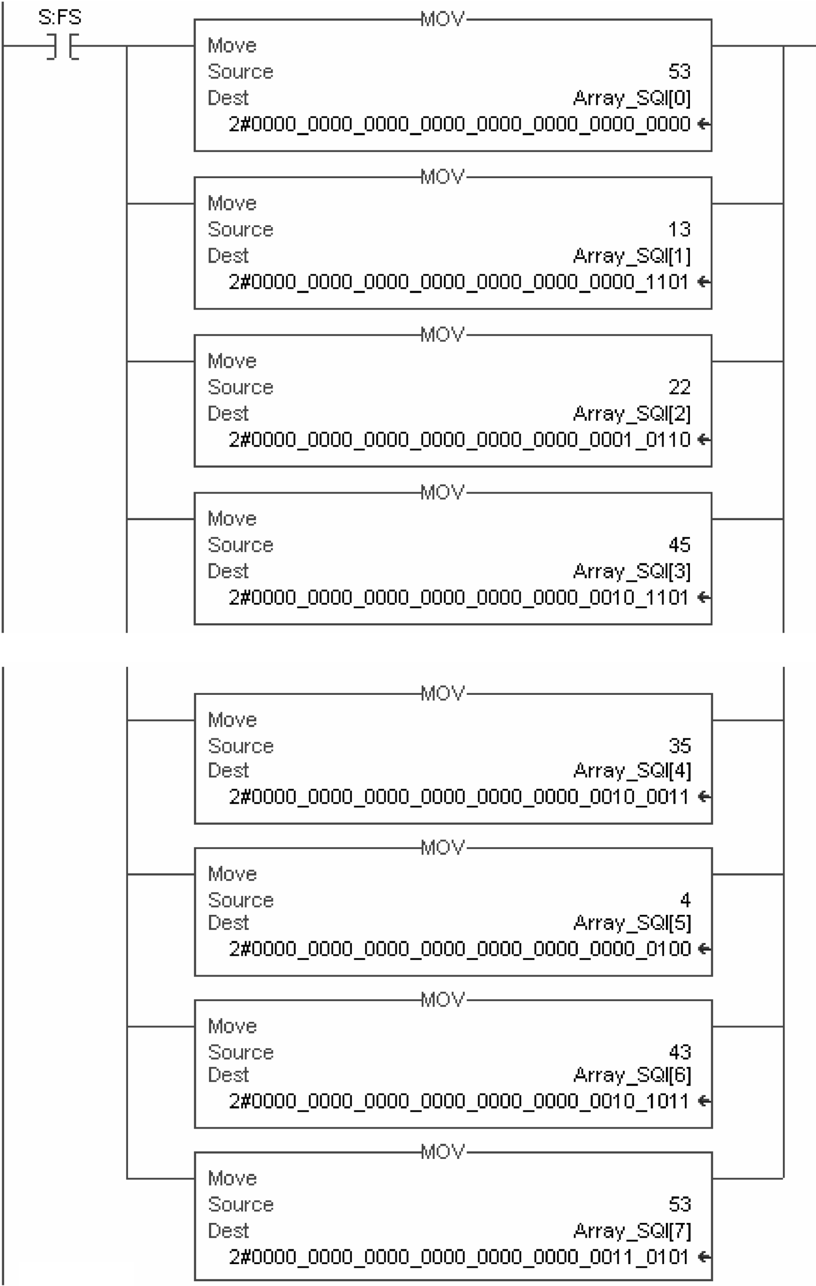


图 10-24　传送顺序器输入数组的设定值

组操作指令中 Length 项键入的数字都是指的数组长度，也即操作长度，惟独顺序器指令中，这里表示的是顺序器的步长，即它的操作长度。由于还有一个 0 步的存放单元，这个存放单元位于数组的首个数据，如果顺序器步长为 N，实际创建的数组长度应该是 N + 1。

最后，要强调的一点是，不管是例程启动的初始化进入还是调用例程的中途进入，一定要注意将顺序器的 POS 清零，否则指针的残余状态可能带来步状态输出的混乱。如果数组的顺序处理没有 0 步的需求，我建议你采用 FAL 指令，这条指令的递增方式完全可以满足

需求，最重要的是没有 0 步的困扰。

关于 0 步，这里稍作介绍。当数组操作的控制结构数据的指针为 0，顺序器输出指令 SQO 和顺序器装载指令 SQL 的梯级条件如果成立，在例程首次扫描的时候，会执行 0 步的操作，即对数组的第一个元素操作，之后将根据梯级条件的跳变加 1，执行后面的步，待一个循环做完，不再回到 0 步，而是从 1 步进入循环。顺序器的 0 步用来解决初始状态的设定问题，这个过程需要严密的配合关系，容易出错。

顺序器输入 SQI 指令的 0 步，在与顺序器输出 SQO 指令配合使用时，通常就是超始步的状态，只有在最初进入需要，本例中选择起始步和最后步使用相同状态。

顺序器的 0 步原是早期产品特用来解决控制步骤的初始状态的。不管怎么说，顺序器指令是最早的数据文件处理方式，后来的控制器数据处理能力越来越强，数组处理可用的方法越来越多，内存空间也变得非常充足，技巧性强且不容易明了的顺序器指令就用得越来越少了。我们仍然介绍了这组指令，并且分析了它的做法，并不是鼓励大家用这种方式，而是考虑到我们学习编程也是为了增强读程序的能力，有的控制器中的例程还保持了这样的梯级逻辑，也许是系统升级带过来的，确实不是很容易读懂。

第 11 章

程序控制指令的运用

Logix 控制器的连续任务是代码执行的基本循环，期间被周期型任务和事件触发型任务中断运行，然后返回中断点，每个任务中的程序按排列顺序执行，每个程序都有一个主控例程，作为程序启动，主例程调用其他例程的执行，所有的例程则执行梯级逻辑的扫描。例程的扫描遵循从上到下从左到右的原则，从上到下按梯级顺序扫描，从左到右执行梯级的指令。这些就是代码执行的顺序和规则，一旦执行程序控制指令，便将改变扫描的顺序，使得指令执行转移到别处。Logix 控制器指令系统有一套被称为程序控制的指令，如下所列：

- JSR　SBR　RET 调用子例程指令　带入参数指令和带回参数指令；
- JXR　调用外部子例程指令；
- EVENT　事件触发任务调用指令；
- JMP　LBL　跳转指令和标号指令　成对使用；
- FOR　循环执行例程指令；
- BRK　终止循环执行例程指令；
- MCR　主控复位指令；
- UID　UIE　进入中断禁止指令和解除中断禁止指令；
- TND　暂停指令；
- AFI　恒假指令；
- NOP　空操作指令。

这些指令，有的是必不可少的操作，有的是用来强调某种操作，有的被用来调试例程，有的具有标志意义，准确地了解它们的作用，才能恰如其分地运用它们。

11.1　调用子例程指令 JSR 及 SBR 和 RET

调用子例程指令 JSR 是使用最为广泛的指令，几乎所有的主例程都要使用 JSR 指令来启动例程或安排执行顺序。如图 11-1 所示的梯级逻辑就是最常见的根据条件执行的简单调用。

如果是调用初始化的例程，则采用的梯级条件是 S: FS，梯级逻辑编写如图 11-2 所示。S: FS 是关键字，只有在控制器进入运行第一次例程扫描时为 1，其余时候均为 0，只有在例程运行的第一次才会执行，一般用来完成初始化的设置操作。

子例程也可以是无条件调用，那就意味着每次扫描都有调用，这样的调用通常被编写在主例程中，如图 11-3 所示的梯级逻辑。

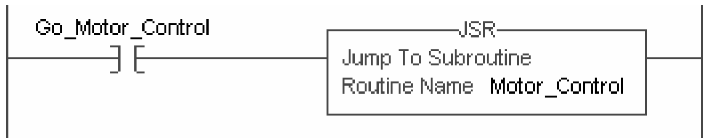


图 11-1　有条件的例程调用

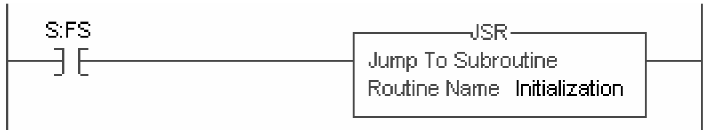


图 11-2　初始化例程的调用

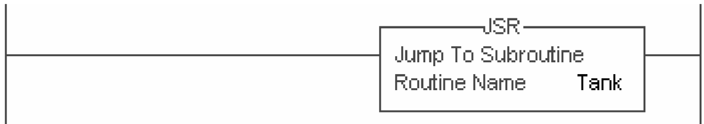


图 11-3　无条件的例程调用

JSR 指令使用简单，但仍有人对在被调用的子例程中是否要使用 SBR 和 RET 感到困惑。子例程的调用分两种情况，一种是简单地调用执行，不带入任何参数，也不带出任何参数，如前面所示的三种情形，此时在被调用的子例程中不需要编写任何 SBR 和 RET 的指令，只需编写与执行有关的梯级逻辑便可。

另外一种情况是要带入参数、带出参数或带入带出参数，如图 11-4 所示是调用例程的 JSR 指令，指令中要指定带入带出的参数的数据标签，带入的数据标签将与被调用例程 SBR 指令中指定的参数数据标签交换数据；带出的数据标签在被调用例程执行完毕后，将与最后一级梯级的 RET 指令中指定的数据标签交换数据。



图 11-4　带入带出参数调用例程的 JSR 指令

在被调用的子例程中，需要带入参数的要编写 SBR 指令，且要放置在被调用子例程第一个梯级的最前面；需要带出参数的要编写 RET 指令，且要放置在被调用子例程的最后一个梯级；需要带入带出参数的则 SBR 指令和 RET 指令都要编写，如图 11-5 所示的梯级逻辑。

一般来说，无条件调用的子例程总是连续不断地执行的，不会有数据残留的问题需要关注。有条件调用子例程的执行则需要考虑数据的状态，当子例程停止调用时，梯级逻辑运行结果修改的数据会保留不动，尤其是非保持性指令值得探讨，例如输出位逻辑指令 OTE 会保留它被修改的原样，只要例程没有再次被调用，这个位状态就会一直保持，这和 MCR 等指令执行后的状态是不一样的，它不会自行复位。为了避免子例程执行受到前次执行残留数

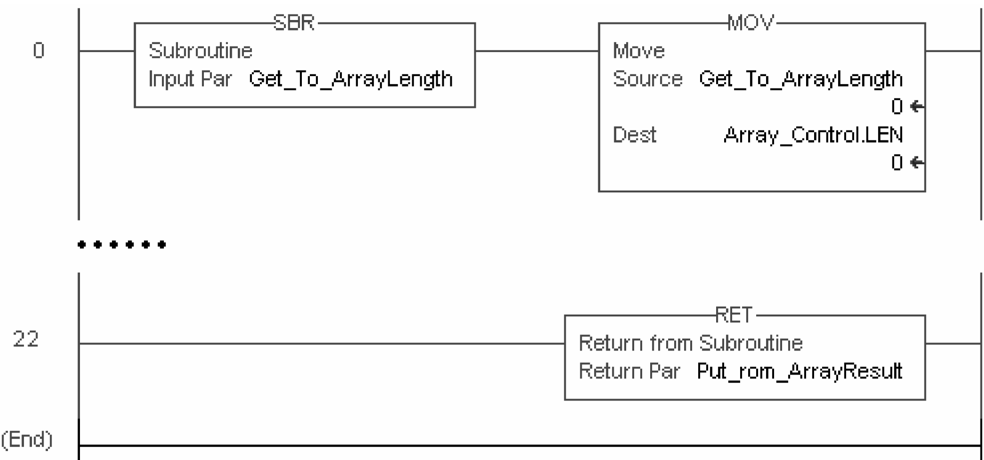


图 11-5　被调用子例程中的有关带入带出参数的梯级逻辑

据的影响，有的满足条件才调用的子例程编写了进入时清除残留数据的初始化操作的梯级逻辑。

像 TON 这样的指令就要非常关注了，通常的计时器启动都会有自动复位的作用，只有子例程的调用是不会清除原有的累加值的。更重要的是，每当例程调用，TON 指令被扫描时会修改累加值 ACC，它会将上次扫描到这次扫描的时间间隔迭加在累加值 ACC 上，如果子例程调用的时间间隔超出了计时器可容纳的时间间隔范围，这时计时器的 ACC 值就变得不可靠了。Logix 控制器是 32 位的数据处理单元，它预留了 22 位来存放这个间隔时间值，可以算出间隔时间最大值是 69. 905min，当一个含有计时器指令的例程被停止调用 69. 905min 以上，这个计时器的累积值就不能使用了。所以，计时器的处理在这种情况下须要谨慎。

同样是调用子例程的指令 JXR 是调用外部子例程的指令，这条指令仅仅在 SoftLogix5800 的软控制器中才会用到，因为只有在运行着 SoftLogix5800 软件的同一台计算机中，才有可能访问到第三方的程序，这里就不举例说明了。

11. 2　事件触发任务调用指令 EVENT

事件触发任务调用有为数不少的触发源，其中之一就是指令调用，事件触发调用指令 EVENT 和调子例程指令 JSR 究竟有何相似和不同之处呢?

它们的相似之处在于，最终都是执行一段梯级逻辑。不同之处在于，JSR 指令只能在程序范围内调用，EVENT 指令却可以在控制器的全局范围内调用，位于任何一个例程中的 ENVENT 指令一旦被触发便立即执行调用任务。还有一个不同之处是 JSR 指令调用的例程在执行之后不会进行输出的处理，EVENT 指令调用的任务在执行之后可以进行输出的处理。

如图 11-6 所示，是一条关于调用系统关闭事件任务的梯级逻辑，这样的调用执行可以放在控制器项目所有的例程中，随时触发调用。

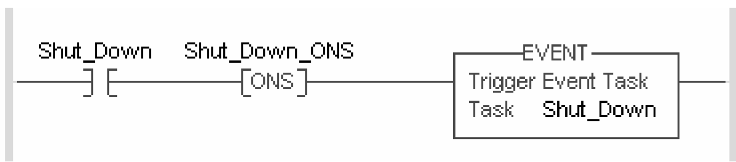


图 11-6　调用系统关闭事件任务的梯级逻辑

类似的运用还有控制器全局的初始化操作，也可以用事件触发任务调用的指令来完成。事件触发任务调用较之其他的触发源，响应不是及时的，只有在指令所在梯级被扫描，指令执行时才完成调用。

11.3　跳转指令 JMP 和 LBL

跳转指令的作用是在同一个例程中，变换梯级扫描的顺序，在满足梯级条件下，跳转指令 JMP 启动跳转动作，与之成对的 LBL 标号指定了跳转进入的梯级。

跳转指令 JMP 和标号指令 LBL 是成对执行的，跳转可以往前跳，根据梯级条件跳过一段梯级不予执行，接着标号之后的梯级继续扫描，这和 MCR 指令的运用有些相似，但不同的是没有后扫描的处理，不会对非保持型指令做复位操作。

跳转也可以往回跳，构成循环执行。我们在数组指令章节中介绍的 FAL 所完成的初始化工作是非常简单直观的，在没有 FAL 指令的情况下，我们就要用跳转指令的循环作用来完成了。如图 11-7 所示的梯级逻辑就是这样一个实例，这也是一种经典的用法，在传统产品的程序中常常可以遇到。

梯级逻辑要完成的初始化工作是：

- 清除实数数组（10 个元素）；
- 清除双整型数组（20 个元素）；
- 清除计时器数组 ACC 值（20 个元素）；
- 清除计数器数组 ACC 值（15 个元素）。

例程的第一次扫描，S: FS 常闭位输入指令梯级条件成立。首先，在梯级完成一般的清除工作，对实数数组和双整型数组采用 FLL 指令，充填 0 实现数组清零。同时清除下面梯级将要使用的指针，令指针从 0 开始计数，这个指针将指向清除对象计时器和计数器累加值的元素序号。

下面的梯级进入 A 跳转，这是返回跳转，这段梯级逻辑反复执行 20 次才会离开。每当循环一次，清除指针 Clear_Position 加 1。计数器清零在循环 15 次之后，梯级条件不成立而不再操作，计时器清零工作则继续余下的 5 次循环，直到条件判断指针不满足小于等于 20，梯级条件的不成立，不再执行 A 跳转，接着下面梯级的扫描，从而结束跳转循环。

例程的第二次扫描就因 S: FS 常开位输入指令梯级条件成立，B 跳转直接跳过，执行后面的梯级逻辑扫描，此属于往前跳转的实例。

注意到完成这些工作，此段梯级逻辑用了标号 A 和标号 B 的两对跳转指令。

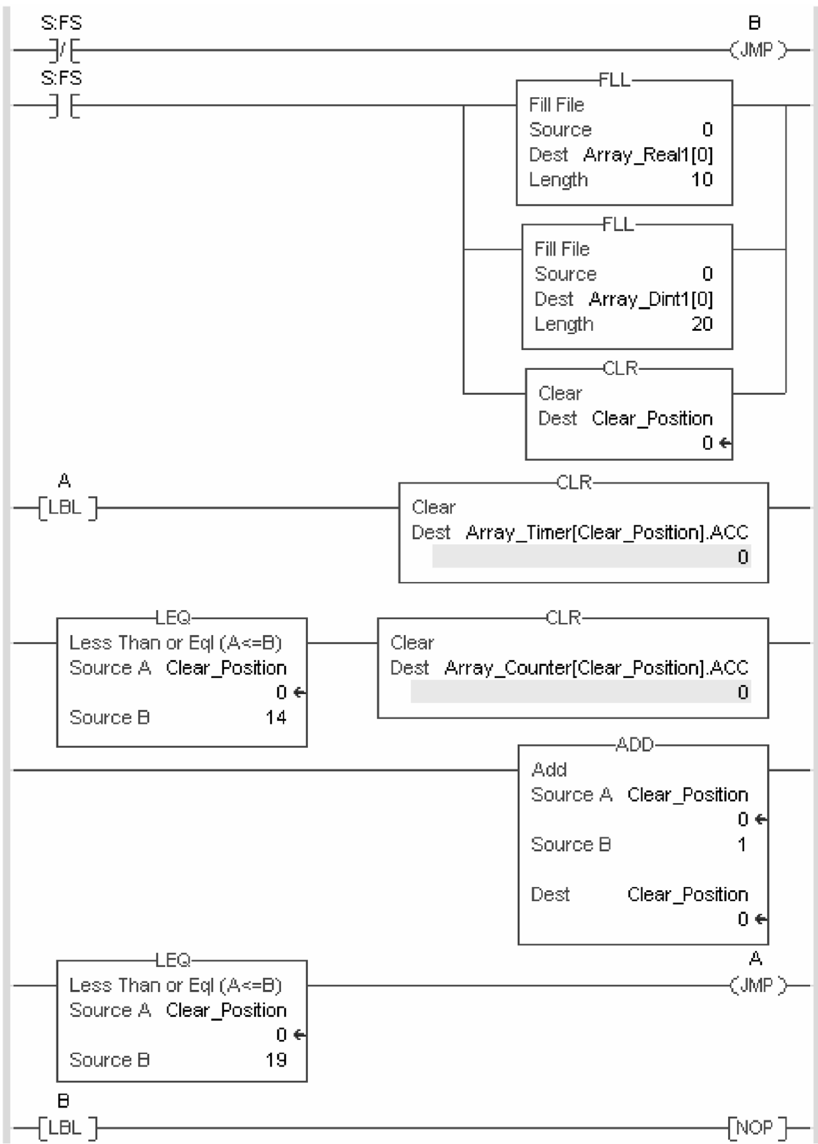


图 11-7　跳转循环清除数组数据

11.4　循环指令 FOR

满足同样的需求，下面试着用循环指令编写梯级逻辑来实现。如图 11-8 所示的梯级逻辑，循环指令 FOR 的执行将调用初始化子例程 Initialize，在 FOR 指令中设定初始量为 0，终止量为 19，步长为 1，循环指令将连续调用 20 次才结束。

在初始化例程中编写的梯级逻辑如图 11-9 所示，一个梯级用 FLL 指令清除数组 Array_Real1 和 Array_Dint1 的元素，计数器数组 Array_Counter 元素 ACC 的清零在 15 次以后停止，计时器数组 Array_Timer 元素 ACC 的清零在 20 次以后完成，同时循环调用例程结束。

循环指令可以直接完成跳转指令往回跳转而实现的循环，它用调用子例程的起始量、终

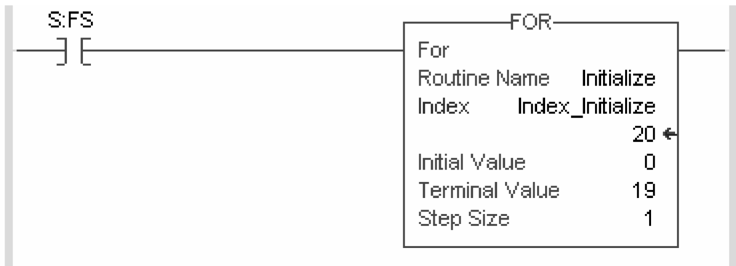


图 11-8　循环调用初始化例程

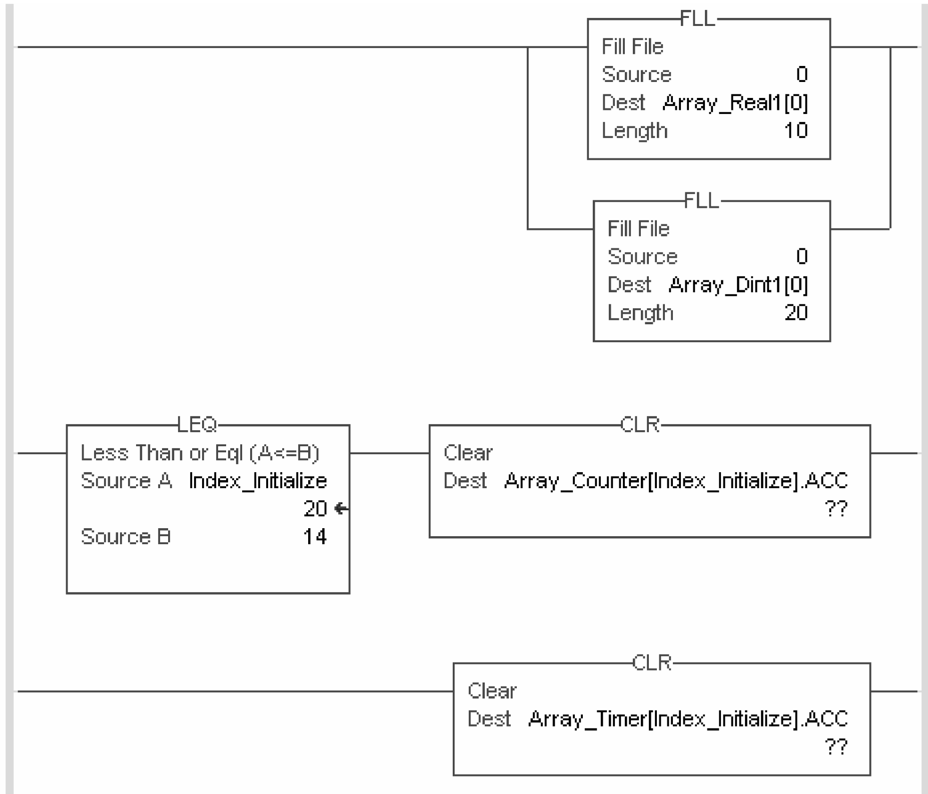


图 11-9　在初始化例程 Initialize 中编写的梯级逻辑

止量和步长来设定循环，灵活而机动。被操作的计时器数组和计数器数组的指针借用了循环的指针变化，无需额外编程修改指针和判定结束，因为这些工作循环指令都是自动完成的。

当没有达到设定的循环次数而需要中断操作，可以用运行在循环例程中的 BRK 指令来中断循环，BRK 指令的梯级条件就是中断条件，当中断执行时，将退出循环操作。

现在，让我们回顾在第 8 章中关于日期时间的 ASCII 码处理，那样的操作执行只处理了一条日期时间记录，如果需要处理一批日期时间记录，就要借助于循环调用来实现了。如图 11-10 所示的梯级逻辑，编写在例程 To_ASCII 中，除了我们熟悉的数据处理过程，在最前面的一个梯级装入将要处理的日期时间，数组元素序号为间接寻址的循环偏移量 DateTimeIndex；在最后一个梯级将处理完毕的日期时间装入数组的单元，数组元素序号亦为间接寻址的循环偏移量 DateTimeIndex。

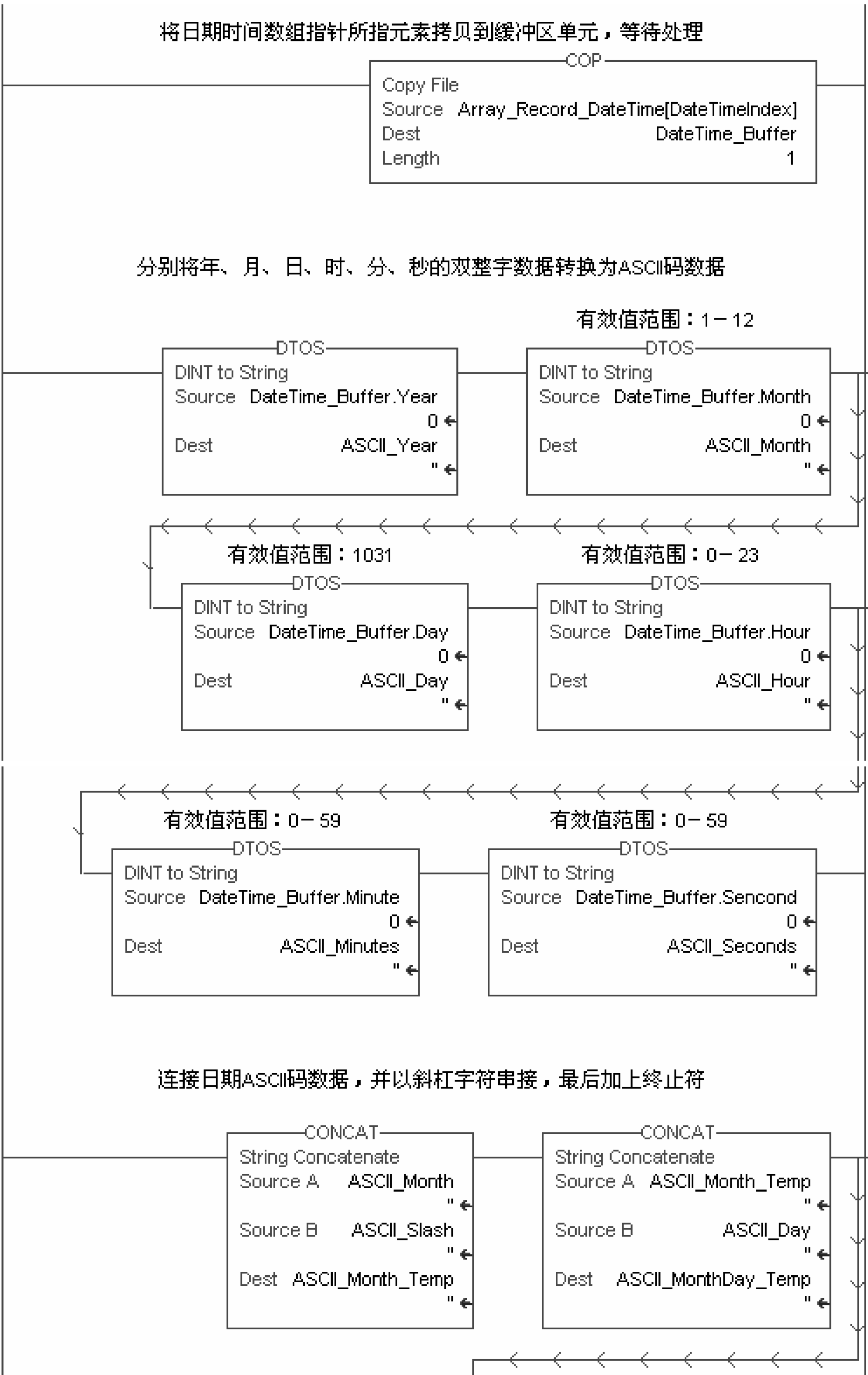


图 11-10　被调用的 ASCⅡ码处理例程

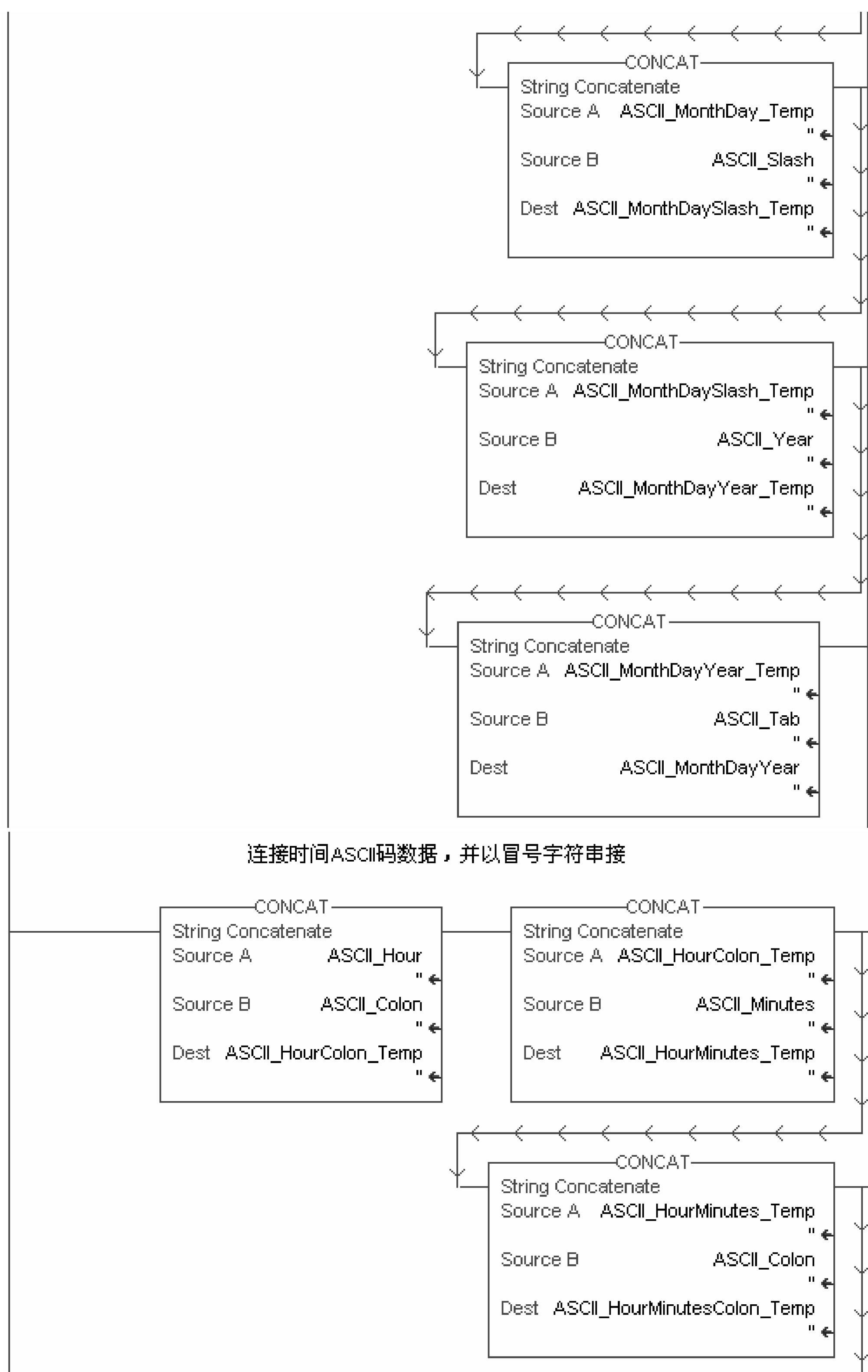


图 11-10　被调用的 ASCⅡ码处理例程（续）

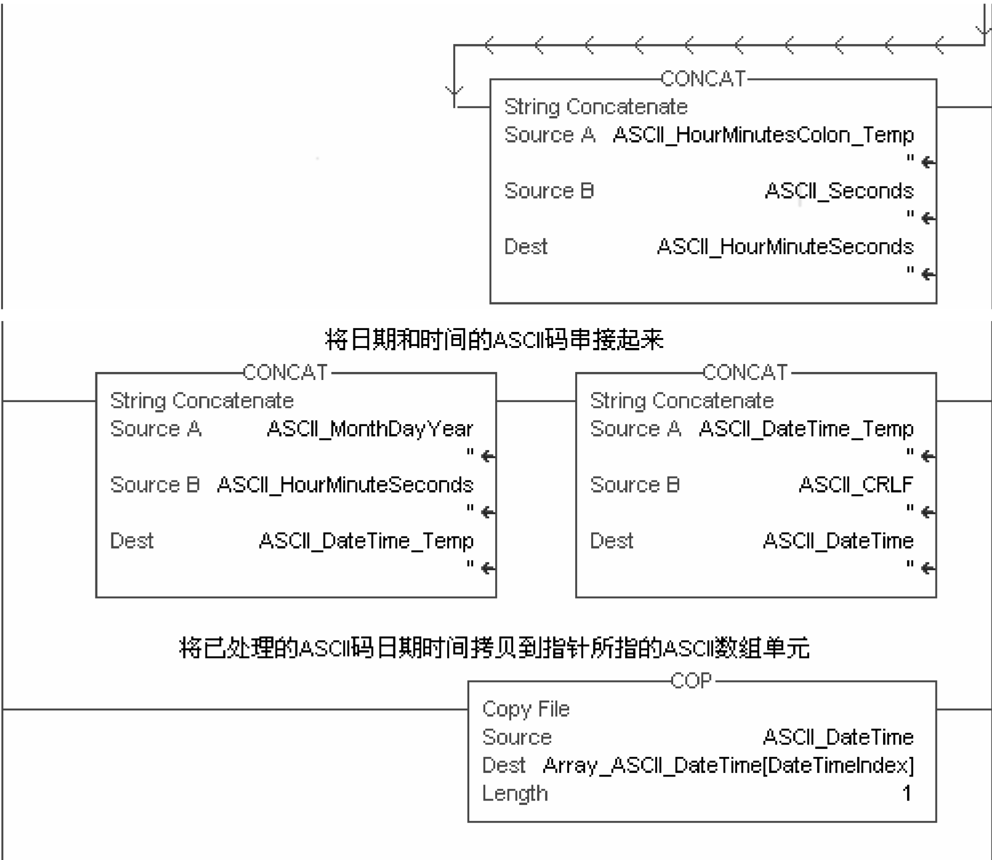


图 11-10 被调用的 ASCⅡ码处理例程（续）

在主例程中编写循环调用指令，如图 11-11 所示， 当梯级条件 DateTime_To_ASCII 成

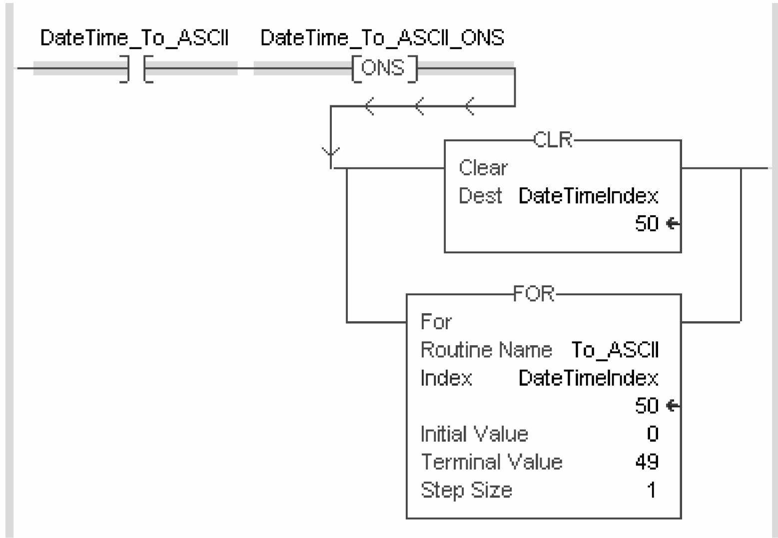


图 11-11 在主例程中编写循环指令调用 TO_ASCⅡ

立，首先执行清除 FOR 指令偏移量 DateTimeIndex，以便重新开始新的循环调用，然后 FOR 指令将调用 To_ASCII 子例程 50 次，完成日期时间数组 50 个元素的转换工作，ONS 指令则确保指令只执行一次。FOR 指令中的偏移量已作为被调用例程中数组的元素序号的指针。

11.5　主控复位指令 MCR

MCR 指令直接的意思是主控复位指令（Master Control Reset），但实际的意义是区域控制复位指令，这里所指的区域是例程中的某段梯级逻辑，早期控制器的程序结构没有现在的控制器产品这样灵活多变，不得不通过 MCR 指令对一段梯级逻辑来进行特殊处理。MCR 指令成对使用来创建一个区域，有条件进入，无条件结束，它的作用有点像括号，用起始 MCR 指令和结束 MCR 指令界定了一段梯级逻辑。

当 MCR 区域梯级条件成立，MCR 区域控制启动，对本区域的梯级逻辑扫描，执行里面相关的逻辑动作。区域里所有的梯级扫描与正常梯级扫描无异。

当 MCR 区域梯级条件不成立时，在 MCR 区域梯级扫描解除，离开本区域控制时，会实行后扫描，这个后扫描所做的事情就是令所有的梯级条件不成立，这将使得非保持型指令复位，这就是区域控制复位指令的复位含义，如果这段梯级逻辑中有 OTE 和 TON 等指令，一定要分析它们的状态及受到的相关影响。

我们用一个实例的执行来了解 MCR 指令的作用。假定有一个这样的需求，一个外部的输出点，当条件成立时，打开 5s，关闭 3s，交替执行；当条件不成立时，输出处于关闭状

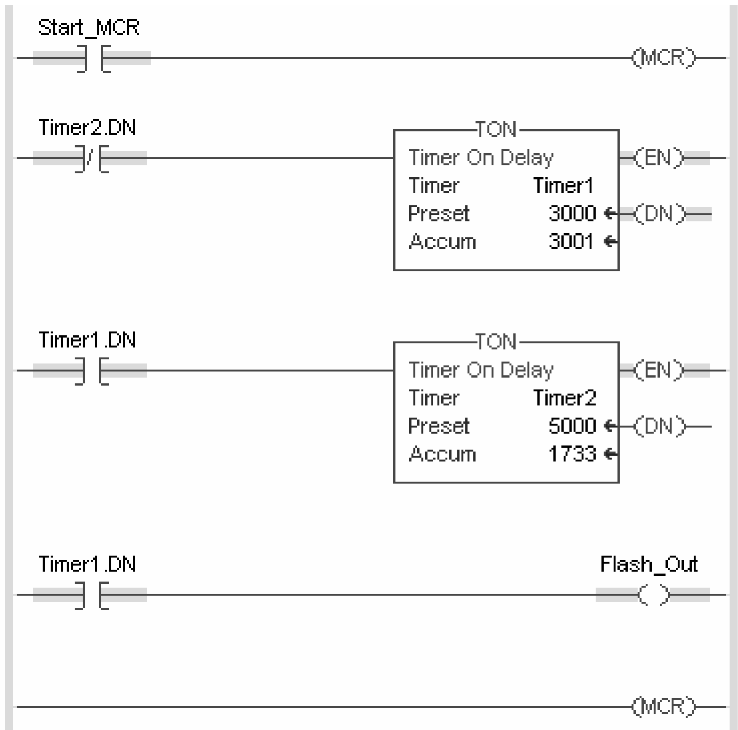


图 11-12　MCR 控制区域激活时的正常运行状态

态，编写满足需求的梯级逻辑。

运行测试，当 Start_MCR 置位，MCR 梯级条件成立，MCR 控制区域启动，区域的梯级逻辑开始执行，计时器 Timer1 和 Timer2 开始计时。Timer1 数据标签的完成位 DN 置 1 的时间可维持 5s，即 Timer2 的计时时间，此时输出 Flash_Out 被接通，直到 Timer2 计时结束，其完成位的常开输入指令将 Timer1 复位并重新开始计时，复位后的 Timer1 的 DN 位复位，输出 Flash_Out 被关断，维持 3s，直到 Timer1 完成位置位。输出点 Flash_Out 将接通 5s，关闭 3s，交替重复运行，运行状态如图 11-12 所示。

当 Start_MCR 复位，MCR 梯级条件不成立，MCR 控制区域关闭，后扫描令 MCR 区域内的所有梯级条件不成立，两个计时器均复位，输出指令 OTE 复位，输出处于关闭状态，这正是我们所需求的，关闭状态如图 11-13 所示。

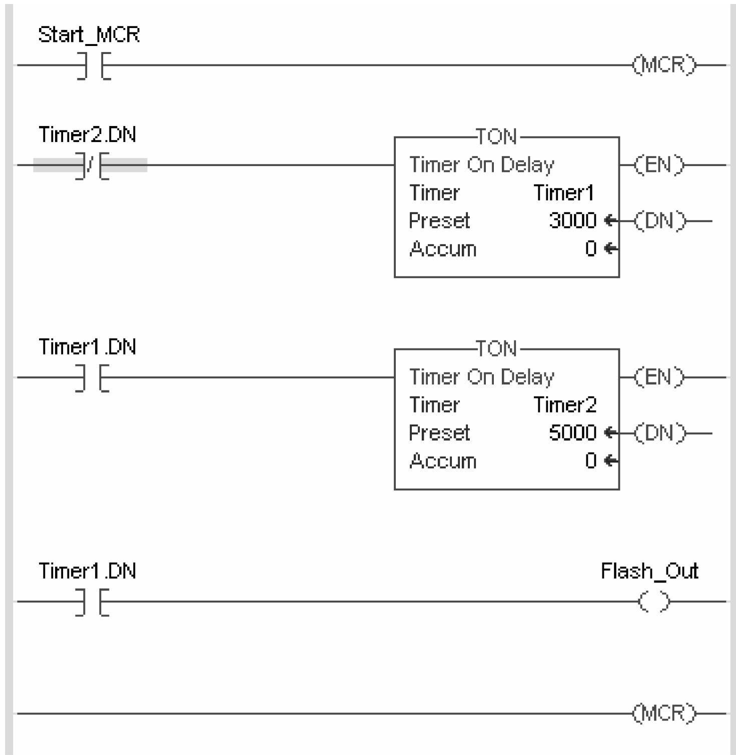


图 11-13　MCR 控制区域关闭时的复位状态

对比常规的子例程的调用，我们发现 MCR 对离开控制后所作的处理是不一样的，子例程不会做任何处理，例程被调用执行逻辑修改数据，例程不调用数据就保持原状，MCR 指令的区域控制在离开时通过后扫描对所有的非保持型指令复位，这是独立于用户控制逻辑的系统介入的动作，正是这个系统介入的动作，完成了梯级逻辑扫描结束后想做的善后工作。所以，当例程调用的残余数据状态不会影响结果时，可以采用调用子例程的办法；当执行逻辑结果的残余数据状态需要复位时，不妨选择执行 MCR 指令来实现区域控制。

顺便提及，可以多对 MCR 指令同时使用，可以嵌套，不可交叉，无条件结束的 MCR 指令是无法识别交叉关系的，当嵌套使用时，这种情形有点像大括号、中括号和小括号的关系。

11.6　禁止中断指令 UID 和 UIE

控制器的活动除了有用户定义的执行代码的执行顺序，还有一些优先权更高的系统活动将介入这些执行顺序，产生中断。但是，有的梯级逻辑的执行如果被中断，可能会带来不可预料的后果。禁止中断指令是一对防止梯级逻辑执行被中断的指令，它也像一对括号，界定一段梯级逻辑，使这段逻辑不受干扰，完整地执行完毕。禁止中断指令跟同步拷贝指令 CPS 一样，也是优先权很高的指令，当外部有中断请求时是不会停止界定的这一段梯级逻辑的执行，直到梯级逻辑执行完毕，最后的解除中断禁止指令 UIE 的执行才结束，来响应外部中断。

例如，为保护 FSC 指令执行的查找过程不被外部中断，采用禁止中断指令 UID/UIE 来保护，如图 11-14 所示的梯级逻辑编写。本条 FSC 指令需要查找 10 个数据，在查找的过程中不容中断。

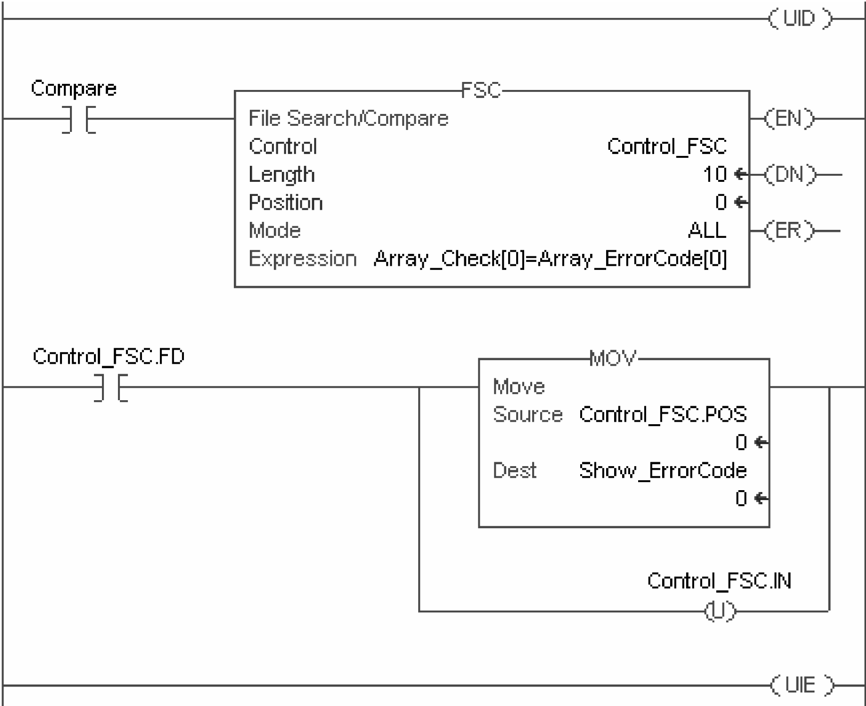


图 11-14　禁止中断的梯级逻辑

11.7　暂停指令 TND

TND 指令是暂停指令，用于例程调试。也许你没有注意到，每个例程的最后都有一个 END 的梯级，如图 11-15 所示。例程梯级逻辑的扫描遇到这个标志就结束了例程的扫描，TND 指令的作用正是如此，例程的梯级扫描一旦遇到 TND 指令，就会在此处停止梯级逻辑扫描，后面的梯级将不会扫描执行。

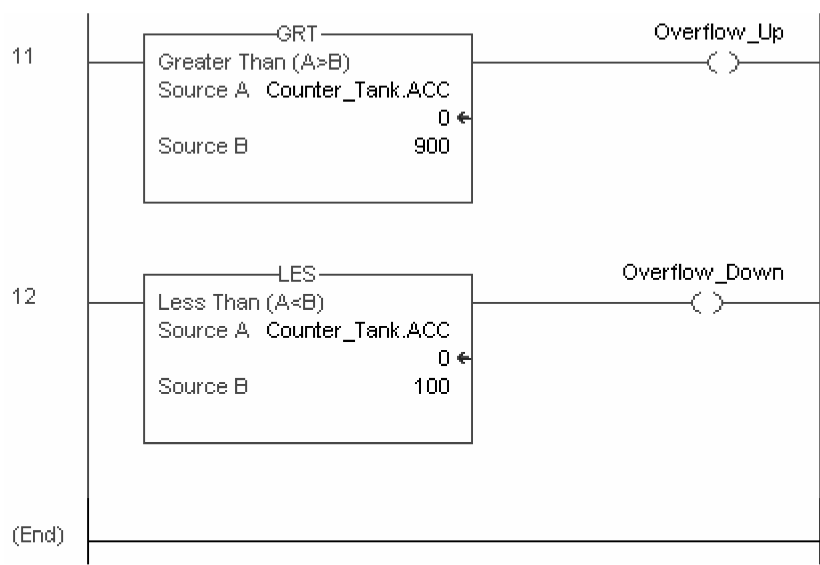


图 11-15 每个例程的最后都有一个 END 的梯级

前面，我们在编写先入先出的堆栈指令采集数据的梯级逻辑的一个应用实例中，曾遇到堆栈数组最后缺少一个数据的情形，一时不大好理解，通过详细分析才找到问题所在，但那只是理论上的分析。如何通过实验来验证其正确性？从而可以放心地使用数组长度加 1 的解决方案。

编写如图 11-16 所示的梯级逻辑，位于 FFL 指令之后的 TND 指令的梯级可以令扫描暂停在 FFL 指令执行之后，在线测试可以看到装载到第 21 个数据时的情形。在数据表中查看，

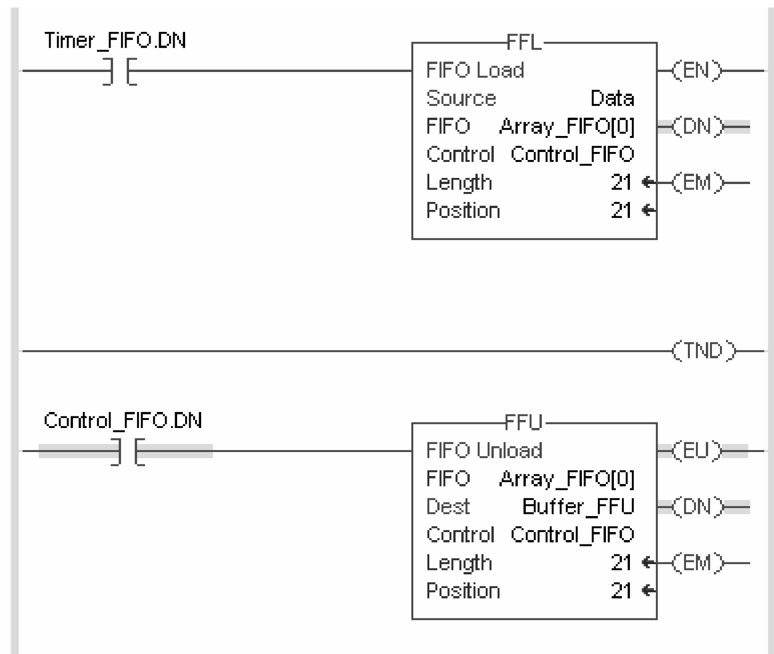


图 11-16 暂停在装载和卸载动作之间

第 21 个数据被装载在最后一个数据单元，由于 FFU 之后的指令都没有执行，FFL 指令和 FFU 指令共用的 Control_FIFO 结构数据的完成位 DN 的梯级条件未作用于执行卸载操作指令 FFU，第 21 个数据暂时地没有消失，前面我们分析过的完成位 DN 置位后执行的瞬间情况得以证实。

用 TND 的指令来分段调试例程也许真的很实用，它可以让例程扫描停止在我们想要停止的地方，使我们能检查和分析应有的状态或结果，用以证实正确与否，但我们一定要明了是如何停止扫描的，这样才能准确地判断和分析测试结果。

11.8　恒假指令 AFI

现场的例程调试还有一条十分有用的指令就是恒假指令 AFI，这条指令一般出现在某个梯级输入的前端，用来否定本条梯级，令该梯级不执行，如图 11-17 所示。调试的时候，如对某个梯级的执行存在某些疑问需要观察，不妨采用恒假指令，停止该梯级输出指令的执行。

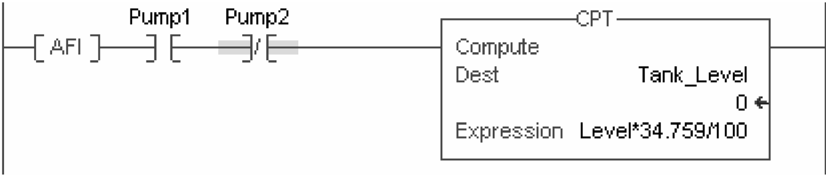


图 11-17　AFI 指令位于某个梯级输入的前端

在没有 AFI 指令之前，工程师们想到了一个巧妙的办法，对串联在一起的常闭输入位指令和常开位输入指令使用相同的位地址，这无疑来自于电路的启发，这样的回路是永远不能通的，如图 11-18 所示的梯级逻辑。这就是后来演变成恒假指令的做法，恒假指令实际上是令梯级条件永远不成立的结果。

这么说来，在梯级的某个输入位指令后面紧跟上一个相反的输入位指令就可以达到 AFI 指令相同的效果，但是这样容易遗忘和混淆，难以在繁多的梯级中一眼看出。恒假指令却是直接的表达，所以 AFI 指令除了否定梯级条件之外，还有标志的意义，在需要梯级条件永远不成立的测试中，建议你使用恒假指令 AFI 而不要使用技巧性的处理。

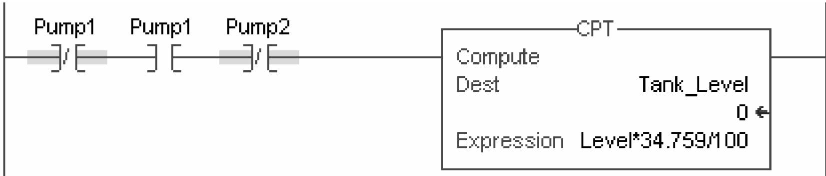


图 11-18　永远也不能成立的梯级条件

以上的测试很容易让人觉得恒假指令 AFI 对所有的输出指令都是令其不执行，以至认为令梯级输出指令不起作用，相当于梯级不存在。其实不然，恒假指令 AFI 对非保持型指令是起作用的，其作用则是复位，你应该没有忘记，非保持型的指令在梯级条件不成立时是复位的，如图 11-19 和图 11-20 所示，这是两个典型的非保持型输出指令，如果在它们的梯级前

面加上 AFI 指令，则令指令永远在复位状态。这样说起来是很明白的，但总有人会出现错误，特别是对待这两条指令时。



图 11-19　非保持型输出指令被复位（一）



图 11-20　非保持型输出指令被复位（二）

曾经有人想对一个输入逻辑关系复杂的梯级的输出点 OTE 指令做测试，用 AFI 将其梯级条件封住，然后在这个梯级的前面另外编写了一条简单逻辑关系的梯级控制这个输出点的 OTE 指令，结果可想而知，输出永远为 0。如果他没有意识到是 AFI 指令的恒假作用的话，将让他得到错误的结论，或者使他对没有得到一个显而易见的逻辑结果而大惑不解。本来在一个例程中同时对一个输出点控制就是应该避讳的事情，加上以为 AFI 指令不会作用于 OTE 指令就更是错上加错了。

对于计时器 TON 指令的类似测试也是如此，如果你想将一个计时器复杂的逻辑输入条件用 AFI 指令封住，然后在另一个梯级用一个简单的逻辑去测试，你会发现这个计时器的累加值永远为 0，根本不会进行时间积累，甚至不论测试梯级逻辑在这个梯级前面或后面，这是因为 AFI 指令所在梯级不断地为这个计时器结构数据标签复位。

再强调一次，梯级恒假并非不作用，AFI 指令是令梯级条件不成立，并非是令输出指令不执行，尽管大多数情况下的确是令指令不执行，一旦涉及非保持型指令，就会产生复位的效果。你不要试图用 AFI 指令恒假一个梯级，同时在另一个梯级测试同一个地址的非保持型指令，得到的结果一定不是你想要的。

千万要记住，任何涉及复位操作的地方，非保持型指令 OTE 和 TON 总是会有特殊的表现，这在预扫描、后扫描、MCR 指令等应用场合都反复提及。

11.9　空操作指令 NOP

空操作指令 NOP 也是很有意思的一条指令，看上去这是一条没有意思的指令，简直毫无意义，什么时候需要空操作呢，都不执行动作，要指令何干呢。

一开始的编程基础一章中，我们曾经提到，输出指令的梯级条件可以是无条件的，没有输入指令只有输出指令的梯级是符合梯形图语法的，但是只有输入指令没有输出指令的梯级是不能被接受的，如果在例程编写中遇到这样的问题，可以用空操作指令 NOP 来解决。

例如刚才我们所做的跳转指令的测试实验，当跳转到标号 B 时，理应接下去进行例程后面的逻辑扫描，此处我们就用空操作来代替，如图 11-21 所示的梯级逻辑。

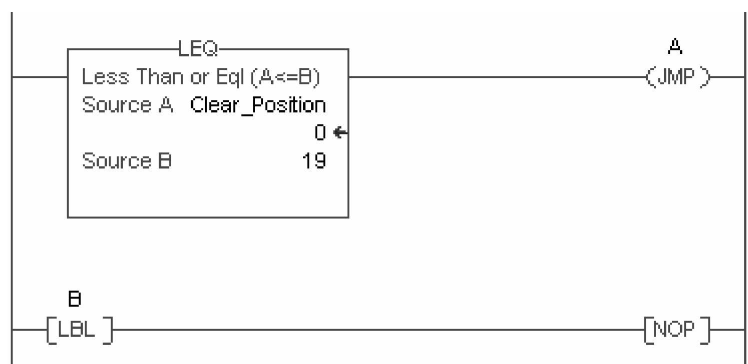


图 11-21　用空操作来代替的梯级逻辑

在接受产品技术培训时，有的编程实验的实例会留下一些梯级空白让我们去填补，那些梯级就用 NOP 指令预留着，我们习惯而迅速地找到它们来完成我们的练习。

毫无疑问，在离线编程或现场调试时，我们编写了合适的梯级条件，一时对执行的动作没有把握，不妨编写一个 NOP 指令暂时放着，而不要编写一个真实的输出指令以至不辨真假，到现场后再针对真实的输出来调试。

空操作指令 NOP 具有强烈的标志作用。

第 12 章

特殊指令的编程

本章将要讨论的特殊指令是一组耐人寻味的指令，这是根据需求开发的应用型的指令，不但完成比较复杂的操作，具有比较完整的功能，其数组运作方式将大量重复操作的指令化繁为简。我们将通过应用实例，看到要用字操作的指令完成的大量工作，耗用大量的梯级逻辑，采用数组处理的操作变得非常简单。本章要谈到的特殊指令是：

- FBC 数组位比较指令；
- DDT 诊断检测指令；
- DTR 数据转变指令；
- PID 比例微分积分控制指令。

数组操作指令 FBC 和 DDT 指令相似而又略有不同，在使用时，要注意它们的差别。它们都是输出指令，将源数组跟参考数组进行对比，一个是对比完后不修改参考数组；一个是对比完后马上修改参考数组。DTR 指令和 SQI 指令是相似的，它们都是输入指令，通过比较得到诊断和监测的结果，从而决定了梯级条件是否成立，但它们的结果逻辑恰恰相反，一个是比较结果相同，梯级条件成立；一个是比较结果不同，梯级条件成立。PID 指令则是梯形图逻辑指令系统中最复杂的指令，涉及面最广，参数最多，需要进行现场的调试，编写这条指令最重要的是理清数据的关系，特别是跟模拟量之间的数据换算关系，也即定标数据范围的含义。

12.1 数组（文件）位比较指令 FBC

位比较指令 FBC 是位对位的进行比较，针对一批位数据来操作，是处理能力较强的数组操作指令，因而指令的参数项较为复杂，如图 12-1 所示的指令界面。指令的执行是将源数组 Array_Dint1 和参考数组 Array_Dint2 中的双整数位对位地进行位逻辑的比较，这是一个寄存器的操作，发现位状态不同时，将对应的结果记录在结果数组 Array_Dint3 中。不同于我们之前学习的数组操作指令，这条指令的参数有 3 个操作数组和两个控制结构数据标签。

3 个操作数组：

- 源数组 比较对象，来自采集对象，是一组变化的信息，此为双整字数组。
- 参考数组 参考量，事先设定的参照信息，是一组固定的信息，与源数组同等长度，共用同一个控制结构体，此为双整数数组。
- 结果数组 记录源数组和参考数组对比不匹配结果的位号，在进行新的一轮比较之前，由于不能确定能否全部覆盖之前的旧记录，最好先清除结果数组，以保证结果

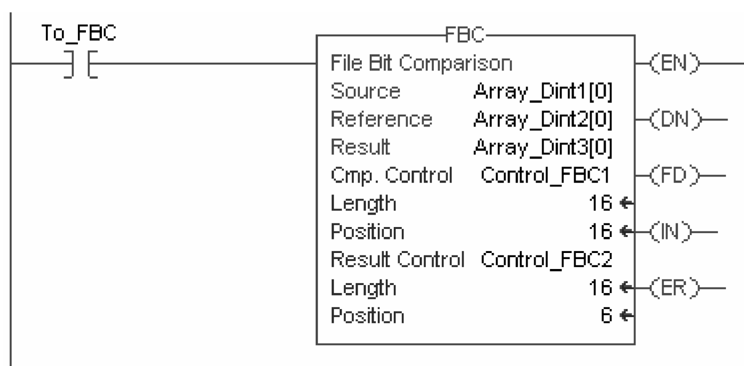


图 12-1　指令界面

可靠。此为双整字数组。

两个控制结构数据标签：

● 比较数组的控制结构数据标签　设定比较的模式，记录比较时的状态和位置，该长度设置指的是参加比较的位的数量而不是数组字的长度。

-每次记录一个不匹配模式　将禁止位 IN 设置为 1。每当梯级条件跳变一次，指令搜索下一个不匹配位，如果发现一个不匹配的位，找到位 FD 置位，并记录不匹配位置，指令停止执行，直到找到位 FD 复位。

-整体模式　将禁止位设置为 0。当梯级条件跳变时，指令执行将所有的位一次比较完毕，如果存在一个以上的不匹配位，找到位 FD 置位，并记录所有的不匹配位 。

● 结果数组的控制结构数据标签　记录不匹配的比较结果，每当找到一个不匹配的

源数组

Array_Dint1[0]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	1	1	0	0	0	1

参考数组

Array_Dint2[0]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	1	1	0	1	0	0	1	0

结果数组

Array_Dint3[0]	0
Array_Dint3[1]	1
Array_Dint3[2]	5
Array_Dint3[3]	11
Array_Dint3[4]	13
Array_Dint3[5]	14
Array_Dint3[6]	
Array_Dint3[7]	
Array_Dint3[8]	
Array_Dint3[9]	
Array_Dint3[10]	
Array_Dint3[11]	
Array_Dint3[12]	
Array_Dint3[13]	
Array_Dint3[14]	
Array_Dint3[15]	

图 12-2　指令运行后数组的数据记录

位，便将该位的所在位置，即 POS 所指的位置值，依次记录下来。该长度一般与比较数组的控制结构数据标签的长度相等，以保证所有的位的不匹配结果都能记录下来。

指令运行结果如图 12-2 所示，将源数组 Array_Dint1 与参考数组 Array_Dint2 进行比较，找到 6 个不匹配的位，顺序将对应的位置记录在结果文件中，请注意结果数组中记录的数值是由 Control_FBC1. POS 提供的位置值。这是采用整体模式运行的结果。

源数组和参考数组都必须定义为双整字数组，即使参加比较的位才 16 位，消耗不到一个双整字，也不能定义为一个双整字，而是应该定义为 1 个字长的双整字数组。在键入地址的时候，一定要键入数组的第 0 个双整字或某个起始双整字，而不是数组的标签名。

结果数组定义为双整字数组，其长度应该与被比较位的个数一致，每个比较位的位置记录都需要一个双整字来表达，在清除的时候，应该采用 FLL 指令来清除。

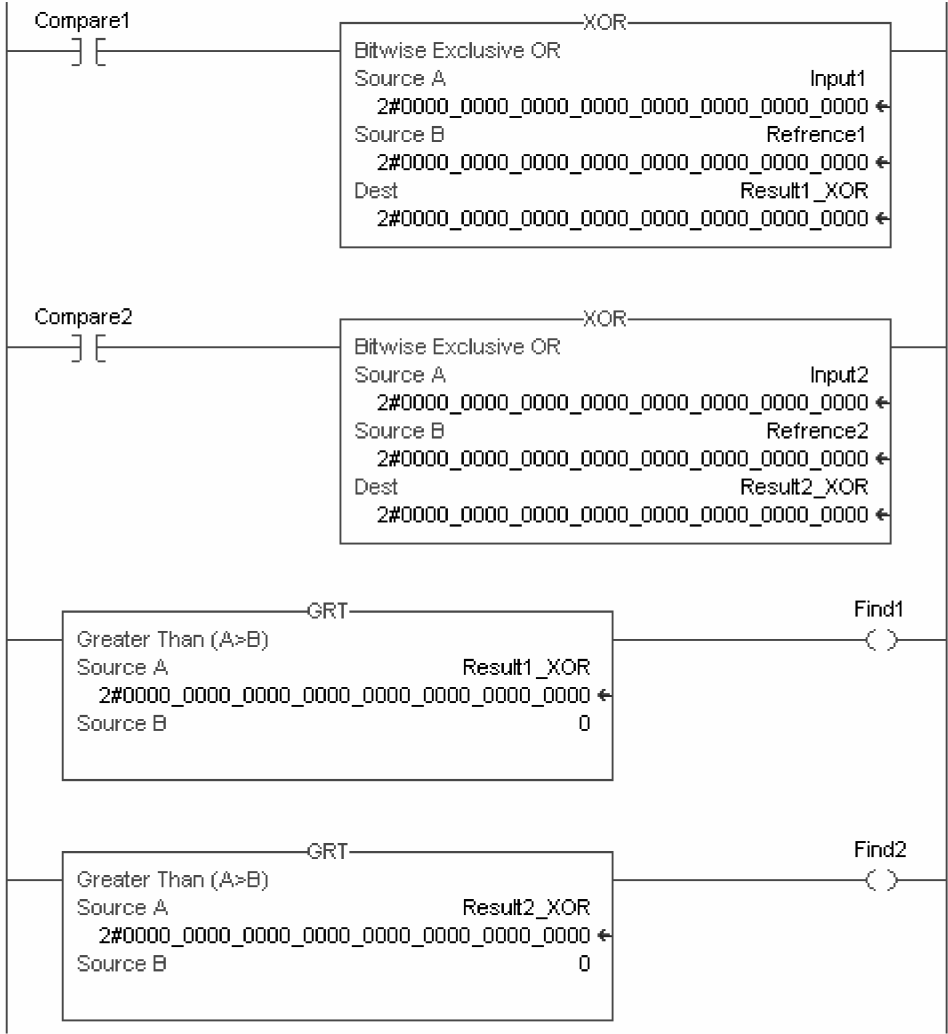


图 12-3　异或运算实例

FBC 指令不同于 DDT 指令，它不修改参考数组，且参考数组的数据是事先设定并固定不变的。

让我们重新审视第 6 章逻辑运算中的异或运算实例，如图 12-3 所示，一共耗用了 4 个梯级才完成了两个双整字的运算，对比结果的尚未进行处理，仅仅能知道双整字的一个以上的位发生了改变，不能对哪些位有变化准确进行判定，否则还需耗用大量的梯级逻辑。如果要运算 10 个双整字，20 个双整字，甚至更多的字，大量的重复性的梯级逻辑无疑是一个负担，且不说增加了控制器的资源耗用，无论是编程人员编写或是维护人员的解读都是异常繁琐。

数组处理的 FBC 指令并不在意数据对比数量的增加，对于这条指令来说，只是要修改一下数组长度。一条 FBC 指令就完成了我们在逻辑运算中的异或运算 4 个梯级的工作，要比较的字越多，数组处理的优势就越发明显，最重要的是，将比较结果整理归档成一个文件，为后续的数据处理提供了简洁而集中的信息。

编写一段梯级逻辑，完成两个双整字 64 位数组的比较，采用整体模式，如果找到一个以上不匹配的位，将比较结果 COP 到数据结果缓冲数组，交给后面的执行代码处理，如图 12-4 所示编写的梯级逻辑。

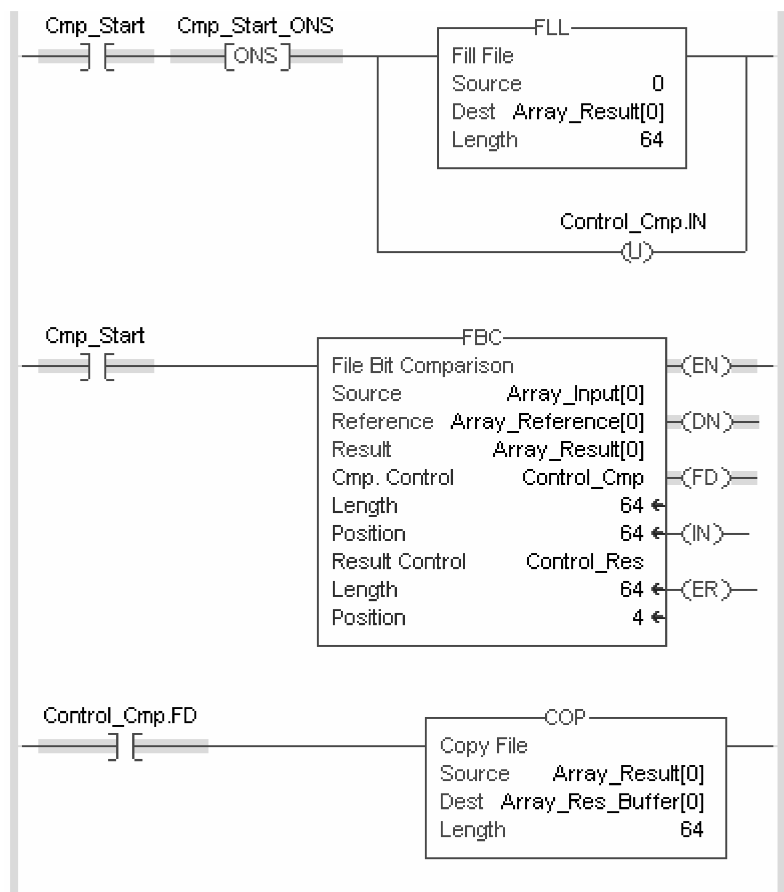


图 12-4　对 64 位进行比较的梯级逻辑

如图 12-5 所示是 FBC 指令执行后在数据表中所看到的数据结果。结果数组所提供的信息可以作为后续梯级逻辑处理的判断梯级条件。

− Array_Input	{...}
+ Array_Input[0]	2#0000_0000_0000_0001_0000_1000_0000_1000
+ Array_Input[1]	2#0000_0000_0000_0100_0000_0000_0000_0000
− Array_Reference	{...}
+ Array_Reference[0]	2#0000_0000_0000_0000_0000_0000_0000_0000
+ Array_Reference[1]	2#0000_0000_0000_0000_0000_0000_0000_0000
− Array_Res_Buffer	{...}
+ Array_Res_Buffer[0]	3
+ Array_Res_Buffer[1]	11
+ Array_Res_Buffer[2]	16
+ Array_Res_Buffer[3]	50

图 12-5　FBC 指令执行后的数据结果

12.2　诊断检测指令 DDT

诊断检测指令 DDT 跟数组位比较指令 FSC 是非常相似的，指令的参数项较为复杂，如图 12-6 所示的指令界面，指令的执行是将源数组和参考数组中的双整字位对位地进行比较，这是一个寄存器的操作，发现不同时，将结果记录在结果数组中，并修改此位，使之与源位一致。非常特殊的是，这条指令有 3 个操作数组和两个控制结构数据标签。

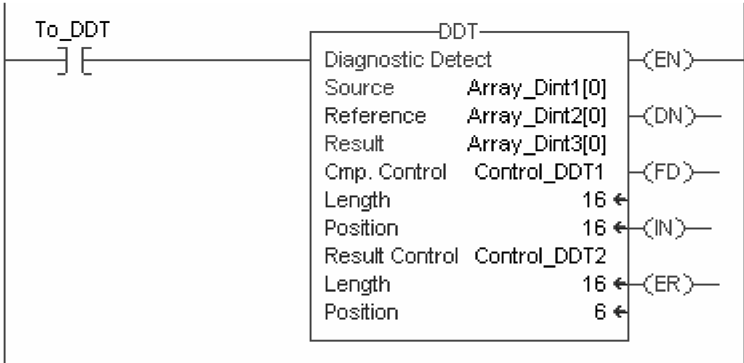


图 12-6　指令界面

3 个操作数组：

- 源数组　比较的对象，是一组变化的信息。此为双整数数组。
- 参考数组　参考量，不断被修改的参照信息，是一组变动的信息。此为双整数数组。
- 结果数组　记录不匹配结果的位号，在进行新的一轮比较之前，最好先清除结果数组。此为双整数数组。

两个控制结构数据标签：

- 比较数组的控制结构数据标签，设定比较的模式，记录比较时的状态，该长度设置指的是参加比较的位的数量。

-每次记录一个不匹配模式　禁止位 IN 设为 1，每当梯级条件跳变一次，指令搜索下一个不匹配位，如果发现一个不匹配的位，找到位 FD 置位，并记录不匹配位置，指令停止执行，直到找到位复位。

-整体模式　禁止位设为 0，当梯级条件跳变时，指令执行将所有的位一次比较完毕，如果存在不匹配位，找到位 FD 置位，并记录所有的不匹配位 。

- 结果数组的控制结构数据标签　记录不匹配的比较结果，每当找到一个不匹配的位，便将该位的所在位置，即 POS 所指的位置值，依次记录下来。该长度一般等于比较数组的控制结构数据标签的长度，以保证所有的位的不匹配结果都能记录下来。

指令运行结果如图 12-7 所示，将源数组 Array_Dint1 与参考数组 Array_Dint2 进行比较，找到 6 个不匹配的位，顺序将对应的位置记录在结果文件中，请注意结果数组中记录的数值是由 Control_DDT1. POS 提供的位置值。这是采用整体模式运行的结果，参考数组的数据已被修改，与源数组的数据一致，等待下一次比较。

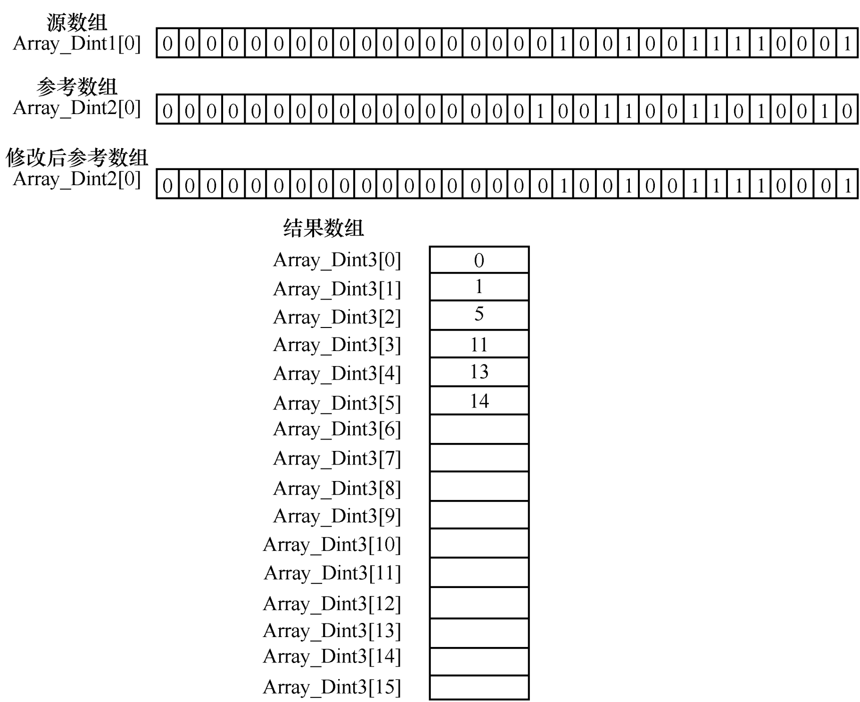


图 12-7　指令运行后数组的数据记录

同样的，在第 6 章中讲到的异或运算不但要获得比较的结果，还要修改参考双整字以得到新的参考量，对每一个双整字都要用一条 MOV 指令传送，梯级逻辑如图 12-8 所示。在需要修改参考量的需求下，例程变得更为繁杂和臃肿。

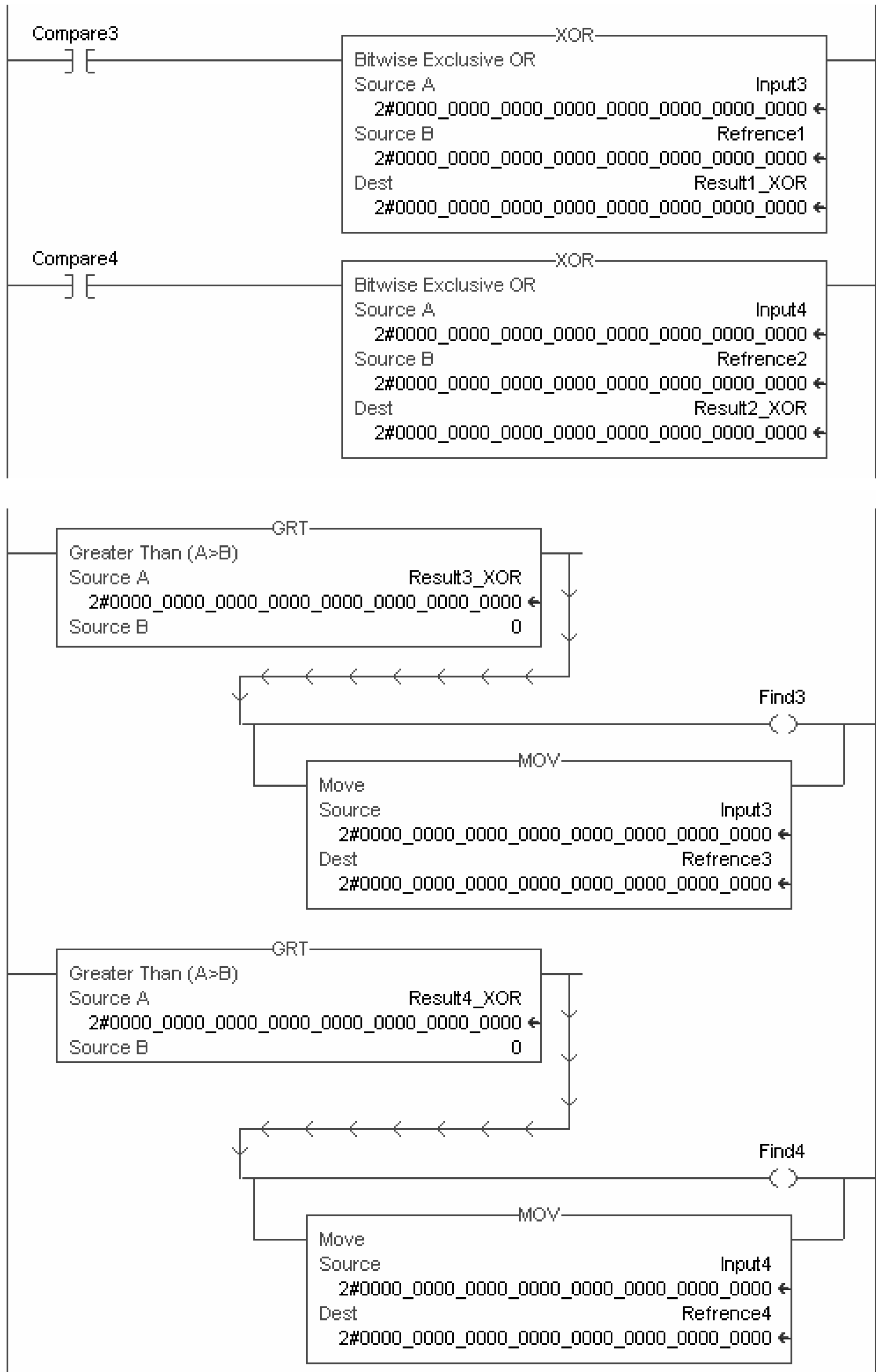


图 12-8　异或运算并修改参考量的实例

如同刚才 FBC 的讨论，对双整字的复杂而繁琐的处理，在数组处理的 DDT 指令执行变得简单易行，针对同样的需求，只要编写如图 12-9 所示的梯级逻辑便可。

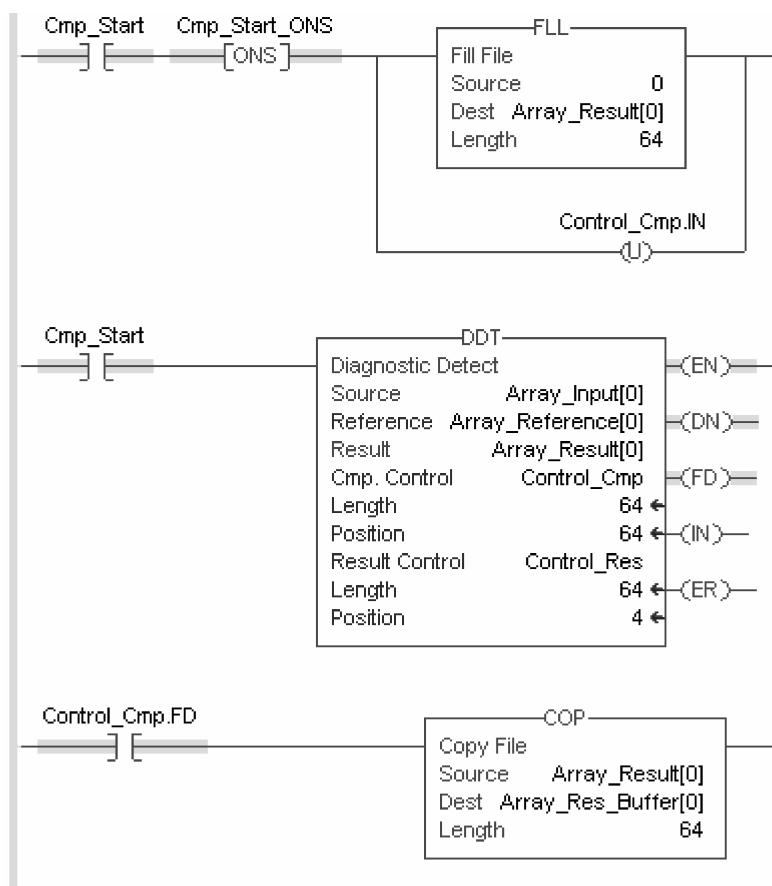


图 12-9　对 64 位进行比较并修改参考量的梯级逻辑

对于需要修改参考量的需求，无需增加更多的梯级逻辑处理，只要选不同的数组处理指

⊟ Array_Input	{...}
⊞ Array_Input[0]	2#0000_0000_0000_0001_0000_1000_0000_1000
⊞ Array_Input[1]	2#0000_0000_0000_0100_0000_0000_0000_0000
⊟ Array_Reference	{...}
⊞ Array_Reference[0]	2#0000_0000_0000_0001_0000_1000_0000_1000
⊞ Array_Reference[1]	2#0000_0000_0000_0100_0000_0000_0000_0000
⊟ Array_Res_Buffer	{...}
⊞ Array_Res_Buffer[0]	3
⊞ Array_Res_Buffer[1]	11
⊞ Array_Res_Buffer[2]	16
⊞ Array_Res_Buffer[3]	50

图 12-10　DDT 指令执行后的数据结果

令便可。如图 12-10 所示是 DDT 指令执行后，在数据表中所看到的数据结果，它与 FBC 指令执行之后的数据结果是相同的，不同的是参考数组 Array_Reference 被改写为与源数组 Array _Input 一致。结果数组所提供的信息可以作为后续梯级逻辑处理的判断梯级条件。

DDT 指令惟一不同于 FBC 指令的是，当源数组与参考数组比较不相同时，它要修改参考数组，这条指令适合用来监视变化的位。运用编程结构几乎不变的梯级逻辑，仅用 DDT 指令替换 FBC 指令，无须增加更多的梯级逻辑处理，较之双整字逻辑运算的处理显得更为简单明了。

12.3　数据转变指令 DTR

这条被称为数据转变的指令听起来跟顺序器指令似乎毫无关系，但作用跟顺序器指令 SQI 几乎一样，它也是一条输入指令，每逢它所在的梯级被扫描，源数组就会和参考数组通过屏蔽进行比较，产生逻辑结果作为梯级条件。比较结果相同，梯级条件为假；比较结果不同，梯级条件为真，并保持一个扫描周期，且将参考值改成与源值一致，等待下一次的比较。

请注意，DTR 指令与 SQI 指令比较结果的逻辑关系是相反的，SQI 指令是比较结果相同，梯级条件成立，还有不同之处在比较结果不同时要修改参考数组，常常用来检测数据是否发生了变化。梯级逻辑编写如图 12-11 所示。

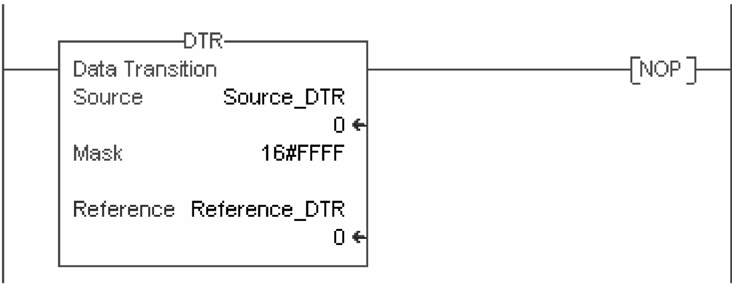


图 12-11　编写的梯级逻辑

指令执行比较的结果有两种情况，相同或不同，如图 12-12 所示。可以看出比较结果相同时，参考值不会修改；比较结果不同时，参考值被修改成跟源参数一样。

该指令适合用于监视一个数据的变化，并在变化时有执行动作，这个动作就是 DTR 指令提供梯级条件的输出指令。请注意，在线运行时编写这条指令存在一定风险，可能会导致输出指令的执行。

我们试着解决一个监视现场数据变化的需求，当这个数据变化时，记录当时的日期时间，并装入一个记录文档，这个记录文档始终保持最后的 50 个日期时间数据。编写的梯级逻辑如图 12-13 所示。

DTR 指令的 Test_Value 标签是监视对象，每当梯级被扫描时，通过屏蔽字与参考量 Reference_Value 标签进行比较，屏蔽字设置为只有两个低字节参加比较，当监视数据发生变化，梯级条件为真，并修改参考量，令其与监视量相同，因此梯级条件只存在一个扫描周期，这足以让 GSV 指令执行获取系统日期时间和 FFL 指令的梯级条件跳变执行。

比较结果相同的情况：

源数值	4	3	2	1	3	5	8	9	
屏蔽字	0	0	0	0	F	F	F	F	
参考值	0	0	0	0	3	5	8	9	本次扫描
参考值	0	0	0	0	3	5	8	9	前次扫描

比较结果不相同的情况：

源数值	0	0	0	0	3	5	8	4	
屏蔽字	0	0	0	0	F	F	F	F	
参考值	0	0	0	0	3	5	8	4	本次扫描
参考值	0	0	0	0	3	5	8	9	前次扫描

图 12-12　指令执行结果的对比

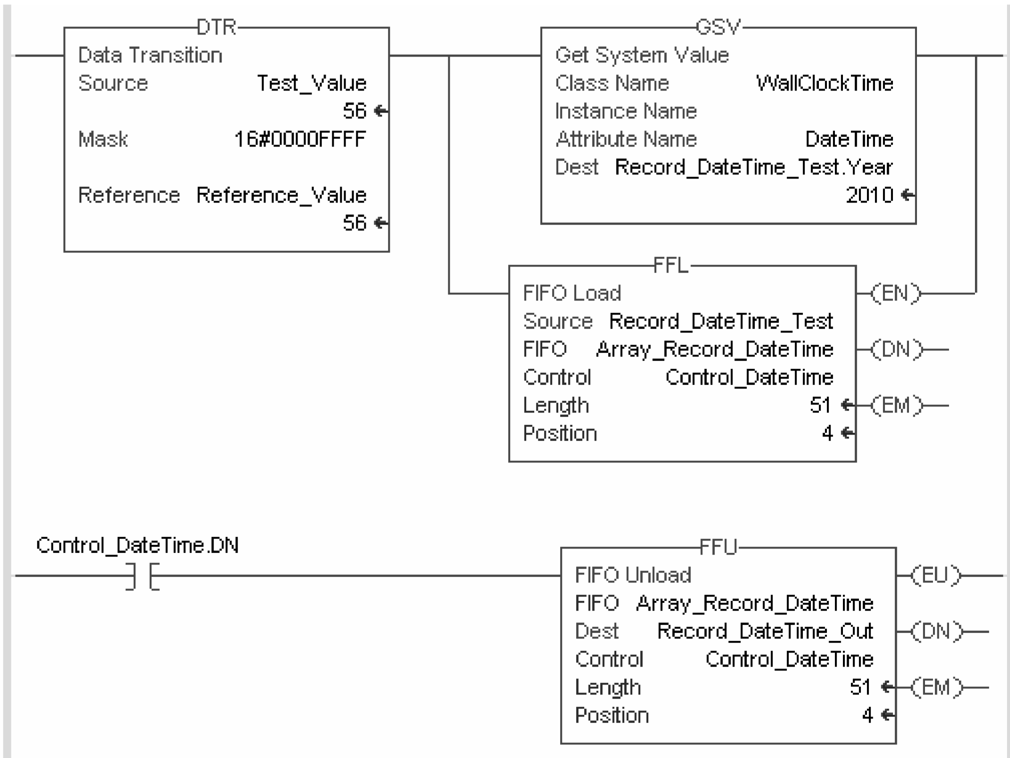


图 12-13　记录数据变化的日期时间并存档

当 DTR 指令给出的梯级条件为真时，GSV 从系统中获取当前日期和时间，同时 FFL 指令装载了这个数据，将其存放在堆栈数组 Array_Record_DateTime 中，管理堆栈数组的控制结构数据标签的完成位 Control_DatrTime. DN 使得 FFU 执行，卸载最旧的日期时间数据，腾出最后一个数据单元来装载最新日期时间数据记录。

装载指令 FFL 和卸载指令 FFU 的配合操作和我们在前面第 10 章中讨论过的相似，不同的是此处的操作对象不是单纯的双整字数组，而是一个结构数据数组，一个用户自定义的日期时间结构数据数组。如图 12-14 所示是此段梯级逻辑运行之后，记录在这个结构数据标签中的部分数据，截取了前面两个元素显示。

⊟ Array_Record_DateTime	{...}
⊟ Array_Record_DateTime[0]	{...}
⊞ Array_Record_DateTime[0].Year	2010
⊞ Array_Record_DateTime[0].Month	11
⊞ Array_Record_DateTime[0].Day	25
⊞ Array_Record_DateTime[0].Hour	1
⊞ Array_Record_DateTime[0].Minute	27
⊞ Array_Record_DateTime[0].Sencond	25
⊞ Array_Record_DateTime[0].Microsencond	248784
⊟ Array_Record_DateTime[1]	{...}
⊞ Array_Record_DateTime[1].Year	2010
⊞ Array_Record_DateTime[1].Month	11
⊞ Array_Record_DateTime[1].Day	25
⊞ Array_Record_DateTime[1].Hour	1
⊞ Array_Record_DateTime[1].Minute	29
⊞ Array_Record_DateTime[1].Sencond	52
⊞ Array_Record_DateTime[1].Microsencond	53684
⊞ Array_Record_DateTime[2]	{...}
⊞ Array_Record_DateTime[3]	{...}
⊞ Array_Record_DateTime[4]	{...}
⊞ Array_Record_DateTime[5]	{...}
⊞ Array_Record_DateTime[6]	{...}
⊞ Array_Record_DateTime[7]	{...}
⊞ Array_Record_DateTime[8]	{...}

图 12-14　记录在结构数据标签中的部分数据

12.4　比例微分积分控制指令 PID

主要控制对象为时序逻辑控制的 PLC，一般过程控制的要求并不是很高，多用于温度、液位、位置的调节，PID 的运用较为简单，一般属于大滞后系统调节，很多情况下都是使用的比例积分控制，略去了微分控制，只追求静态性能。

尽管过程控制的 PID 指令并没有太多的编程技巧可言，但使用并不简单，关键在于清楚每个指令参数的含义和每个设置界面参数的含义，从输入通道经 PID 运算处理直至输出通道的数据形式变更，每一个环节都必须清清楚楚。

如图 12-15 所示的气缸位置控制实例，提供给定量跟反馈量进行比较，产生偏差，PID

指令按偏差运算，产生的控制变量给定气动阀门开启信号，气动阀门开启，气流推动气缸，气缸位置发生改变，位置探测设施提供了反馈信号。反馈信号再次跟给定量比较，产生的误差变小，PID 完成幅度更小的调整，如此反复进行，直到反馈量逼近给定量，误差接近于零，PID 停止调节，气缸位置静止于给定量指定的位置，调节完成。

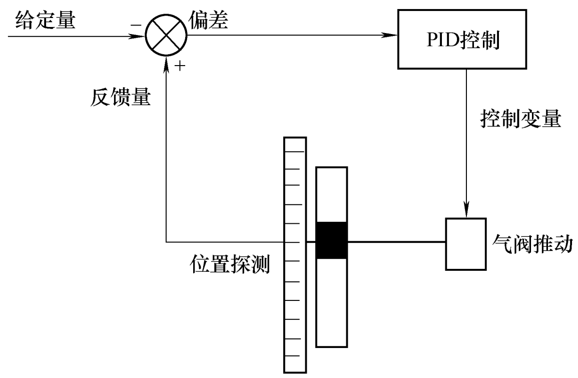


图 12-15　气缸位置控制实例

现在，我们针对这个位置调节系统组态 PID 指令以及测试调节环节。PID 指令完成控制环节的运算，它需要给定量和反馈量，运算的结果是控制量，这就是提供给指令的基本参数，详细的控制要求则是通过指令的组态去实现的。如图 12-16 所示是 PID 指令的面板参数。

一条 PID 控制指令对应一个控制环节，当实行多级控制时则使用多条 PID 指令，主从参数项可选定。本例控制使用一条 PID 指令，即单环控制。

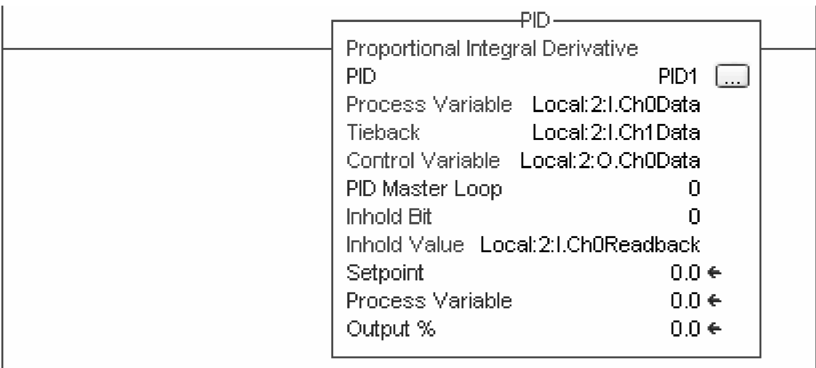


图 12-16　PID 指令的面板参数

PID 指令参数说明：

- 必须给每一条 PID 指令指定一个 PID 结构数据标签，用于存放组态信息和运行状态信息，只有建立了 PID 结构数据标签后，才能点击[...]进入组态页面和调试页面。此例中建立了名为 PID1 的结构数据标签。
- Process Variable：过程变量，从被控现场设备采集的测量数据作为 PID 调节环的反馈量，一般为模拟量输入，指定模拟量模块输入通道数据。此例中选定了模拟量输入模块的 0 通道。
- Tieback：手动控制变量跟随值，来自于手动控制站的手动控制变量一般为模拟量

通道输入，是硬件信号，PID 指令跟随其值，一旦 PID 指令从手动转入自动模式时，从该值开始积分，逐步进入自动控制输出值，实现平滑过渡，防止突跳损坏机械设备。目前一般不会使用这种方式，此例中选定了模拟量输入模块的 1 通道。

- Control Variable：PID 按误差运算的结果，当 PID 在自动模式时，作为 PID 控制环的控制变量，一般为模拟量输出，通过模拟量输出模块的 D/A 转换，模拟量信号送至现场控制设备，不断调节被控设备的控制动作。此例中该输出信号将决定气阀的开度，选定了模拟量输出模块的 0 通道。
- PID Master Loop：多环控制结构中的本条 PID 指令为从环时，输入主环 PID 结构数据标签名称，从而连接了主环的控制参数，当本条 PID 指令为主环或单环时键入 0。此例为单环控制，故输入 0。
- Inhold Bit：保持位，决定输出初始值是否保持在上次的终值上，一般为模拟量输出模块某个通道的保持状态位，这个选择可实现启动的平滑过渡。由于该位与模拟量模块配合使用，此例中不选用。
- Inhold Value：输出初始值保持在上次的终值上，一般为模拟量输出模块某个通道保持量的返回读入数据，输出控制量将从该值开始积分，直至接近控制输出值，从而实现启动平滑过渡。此例中从控制量输出回路模拟量输出模块通道 0 读入。
- Setpoint：PID 指令给定量 SP 的显示值。此为只读数据。
- Process Variable：PID 指令过程变量 PV 的显示值。此为只读数据。
- Output%：PID 控制变量 CV 的百分比显示值。此为只读数据。

PID 指令的参数键入，指定了与 PID 指令有关的外部数据的连接，除了我们熟知的与现场关系密切的反馈量和控制量之外，还有跟随手动变量和初始值保持量，是为保证 PID 指令的运行模式切换和初始启动能够平滑过渡而设置的，可见 PID 指令对启动动作平滑过渡的重视和自我处理能力。

PID 指令参数下部，蓝色箭头所指的是控制环的 3 个监视数据，给定值、反馈值和控制值直接显示在指令面板，以便方便查看，给定值通常是来自于人机界面，由操作员设定，该数据通过换算后直接进入 PID 结构数据标签子元素，不在 PID 指令参数中体现，如本例中 PID1. SP 作为给定对象直接设定。设定值 SP 和反馈值 PV 被要求在同一数据定标范围，这样的比较结果才有意义，控制值 CV 则以百分比表达，可直接观察控制的程度。

PID 指令的组态页面

点击 PID1 的[...]可进入 PID 指令组态和调试界面，选择组态页面，如图 12-17 所示。

PID 组态参数说明：

- PID Equation：PID 控制的运算模式对应不同的微分方程的数学模型，可选择独立增益或相关增益两种运算式，由于比例、微分、积分系数在不同运算模式下有不同的物理意义，所以指令运行时不可更改运算模式。
- Control Action：选择控制方向，SP-PV 或 PV-SP，由此决定了控制环的反馈极性，当系统调试出现正反馈现象，改变选择即改变了反馈极性，得到 PID 调节的稳定系统所需要的负反馈。
- Derivative Of：微分对象的选择，对 PV 进行微分，减少 SP 给定引起的冲击；对偏差微分，获得对 SP 变化量的快速响应。

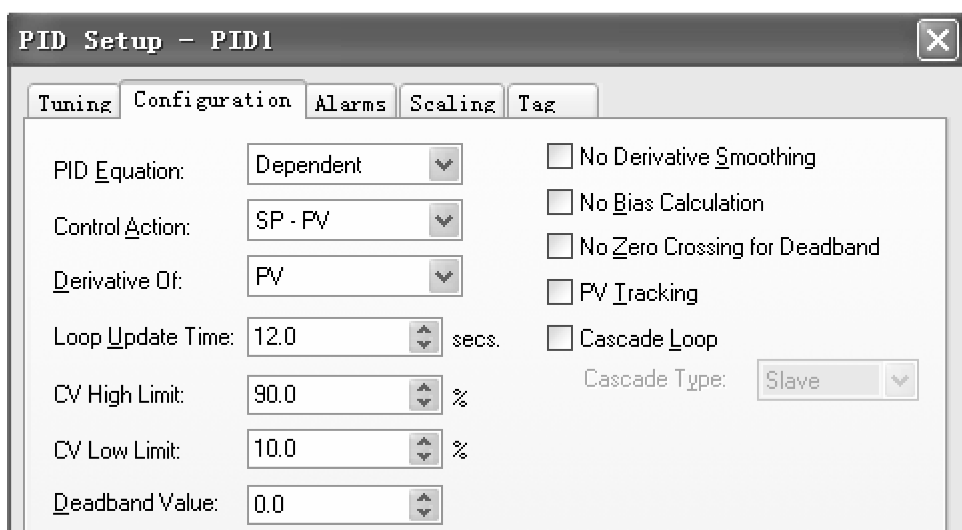


图 12-17　PID 指令组态界面

- Loop Update Time：回路更新时间，不能为零或负数，否则运行时产生控制器次要故障，且指令不执行。这是 PID 运算的周期时间，它与 PID 指令的执行时间有一定关系，如果选择精确中断时间的周期任务调用执行 PID 指令，可令回路更新时间等于周期任务调用时间，且 PID 指令采用无条件梯级。
- CV High Limit：PID 控制变量百分比输出限幅最大值，防止输出正向积分饱和，避免外部执行机构过度正向执行，造成机械损害。
- CV Low Limit：PID 控制变量百分比输出限幅最小值，防止输出负向积分饱和，避免外部执行机构过度反向执行，造成机械损害。
- Deadband Value：死区值，如果硬件需要，可设定过零死区的范围值。
- No Derivative Smoothing：选择有无微分平滑作用。
- No Bias Calculation：选择有无偏置量计算。
- No Zero Crossing for Deadband：选择死区过零或不过零。
- PV Tracking：选择是否进行反馈值 PV 跟踪。
- Cascade Loop：选择回路级联，可选择主回路 Master 或从回路 Slave。

PID 指令的报警组态页面

PID 指令的报警设定是选择报警界限和报警内容，点击进入报警设置页面，如图 12-18 所示。

PID 报警参数页面说明：

- Process Variable［PV］High：过程变量 PV 报警上限值。
- Process Variable［PV］Low：过程变量 PV 报警下限值。
- Process Variable［PV］Deadband：过程变量 PV 死区报警值。
- Positive Deviation：正偏移报警值。

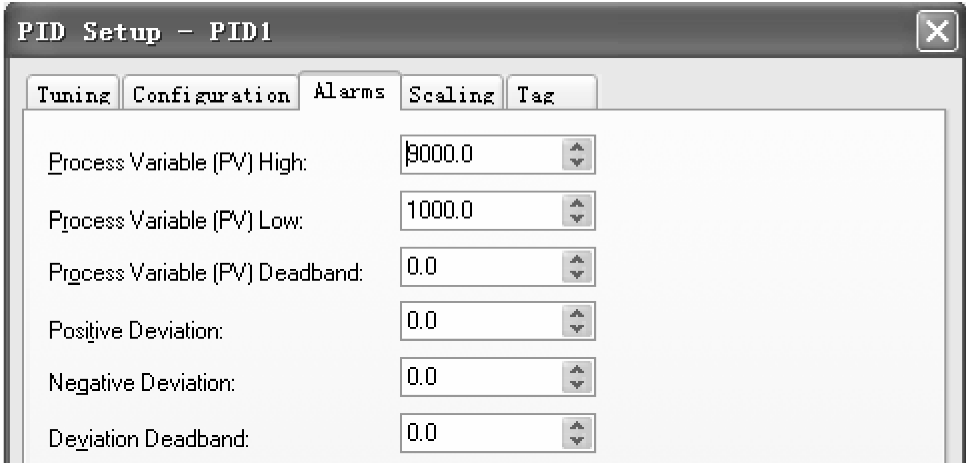


图 12-18　PID 指令报警设置页面

● Negative Deviation：负偏移报警值。
● Deviation Deadband：偏移死区报警值。

PID 指令的定标页面

PID 指令的定标至关重要，是决定 PID 指令能否正常运算的关键。PID 指令的比例、微分、积分运算，经过差分化之后，最终落实在大量的累加和累减运算上，只有固定的运算范围才能获得正确的运算结果，传统控制产品 PLC5/SLC500 都有固定的运算定标范围，Logix 控制器则可由用户指定实数范围以内的数据定标范围，这就是此页面的含义。PID 指令的运用不当，大多数情况是对定标范围的理解有误，不能正确地进行定标，从而导致 PID 运算错误。点击进入定标页面，如图 12-19 所示。

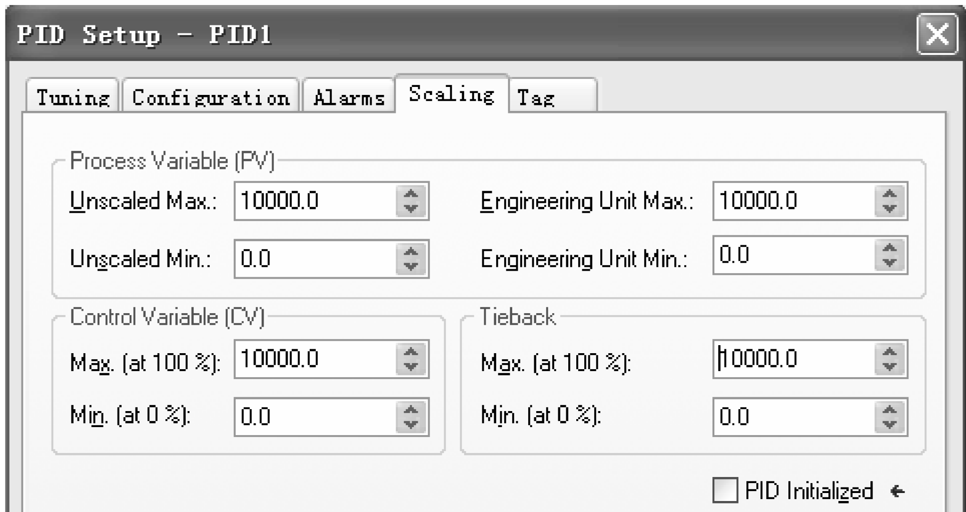


图 12-19　PID 指令定标页面

PID 定标参数页面说明：

● Process Variable（PV）：关于过程变量反馈值 PV 的定标范围，这里也隐含了给定

量 SP 的定标范围，因为这两者是直接相减的关系，只有在相同的范围内相减才有意义，否则由此产生的误差值用于 PID 运算将是不正确的。一般情况下，由操作员设定的 SP 值是控制对象的物理参数，如温度、位置等，或者是百分比，不会是此处给出的定标值，必定要在例程中进行一定的运算换算成此处的定标值范围，再将结果送至 PID 结构数据标签的子元素 SP 值。

-Unscaled Max：一般来自于模拟量模块输入通道的最大定标值，即模拟量模块中的工程定标最大值，此为模拟量数据进入 PID 指令的未定标最大值。

-Unscaled Min：一般来自于模拟量模块输入通道的最小定标值，即模拟量模块中的工程定标最小值，此为模拟量数据进入 PID 指令的未定标最小值。

-Engineering Unit Max：反馈值 PV 在 PID 中再次定标的工程定标最大值，这也是 SP 参照此值确定的输入最大值。

-Engineering Unit Min：反馈值 PV 在 PID 中再次定标的工程定标最小值，这也是 SP 参照此值确定的输入最小值。

● Control Variable（CV）：关于控制变量 CV 的定标范围设定，PID 指令运算的直接结果应该是在如上的 PV 定标范围内，但 PID 指令并不直接给出，而是以百分比的形式给出，让用户转为自己所需要的定标范围，这自然是模拟量输出模块的定标范围，用于 D/A 转换后送至被控设备。

-Max（at 100%）：控制变量最大值，PID 计算结果，SO 百分比最大值所对应的模拟量模块输出通道的工程定标最大值。

-Min（at 0 %）：控制变量最小值，PID 计算结果，SO 百分比最小值所对应的模拟量模块输出通道的工程定标最小值。

● Tieback：手动控制跟踪量定标范围，手动控制跟踪量是当 PID 指令从手动模式切换到自动模式后积分的起始值，此处的定标换算令 PID 指令的起始积分值由此换算而来。

-Max（at 100%）：最大跟踪值，在 PID 中以百分比显示最大值所对应的来自模拟量模块输入通道的最大定标值。

-Min（at 0 %）：最小跟踪值，在 PID 中以百分比显示最小值所对应的来自模拟量模块输入通道的最小定标值。

● PID Initialized：PID 指令初始化选项，此项不选择时，如果在控制器运行时修改如上定标值，可使得重新初始化内部再定标，从而使用新的比例值。

PID 指令把运算定标范围的权利交给了用户，使得用户在使用这个定标范围时具有任意性，这给 PID 运算的精度带来了一定影响。最早的 PID 运算范围是传统产品 PLC5 处理器的 0～4095，这跟当时的模拟量 A/D 转换器的精度有关，其对应的模拟量模块是 12 位的转换精度，PID 运算充分运用了这个精度，后来的 SLC500 处理器的 A/D 转换精度更高，所以它的 PID 指令定标范围在 0～16383，也是充分地利用了模拟量的转换精度。在 Logix 控制器中，一定要尽可能地使用跟 A/D 和 D/A 转换有关的精度，不要急于还原被控对象的物理参数，因为被控对象的物理参数的定标范围都是有限的，远远小过其转换后的数据范围。PID 的运算定标范围，旨在追求运算的精度。

通常从模拟量模块采集的数据，不外乎两种用途，一是用于显示，二是用于运算，前者

可以直接转换为要显示的物理参量，后者则要考虑运算精度，尽可能地使用对应模拟量的转换精度。PID 指令是模拟量常见的运算，切莫辜负了 A/D 或 D/A 的转换精度。

PID 参数调试界面

PID 参数的调节是 PID 性能的调节，点击进入参数调试界面，如图 12-20 所示。

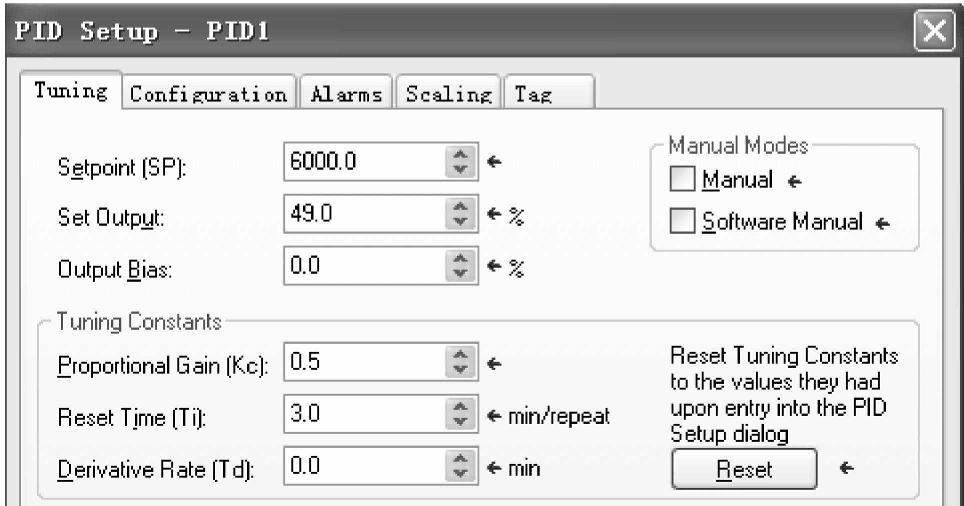


图 12-20　PID 指令参数调试界面

调试页面参数说明：

- Setpoint［SP］：给定值的设定，数值的数据范围必须与 PV 在 PID 定标页面中的再次定标的工程定标范围相同，令其与 PV 值在相同的数据范围进行比较，此值通常在人机界面由操作员以百分比或物理参量的形式操作，要留意编写梯级逻辑给予转换，访问地址是 PID1. SP。
- Set Output：软件手动控制时，在此输入百分比值，此值通常在人机界面由操作员操作，如果不是百分比值直接操作，则要通过梯级逻辑进行转换后进入该值，PID 在自动模式时，则显示输出值。
- Output Bias：输出偏置量百分比值。该值迭加在偏差计算结果上，构成共同的输出 PID1. SO，形成前馈控制。
- Manual Modes：手动控制方式，分为硬件手动控制方式和软件手动控制方式，当两种方式同时被选择时，硬件手动控制方式优先于软件手动控制方式。此为 PID 结构数据标签的子元素，可由例程梯级逻辑改变，或由操作员在人机界面操作改变。

-Manual：硬件手动控制方式，此时 PID 计算结果无效，手动控制站设置的模拟量输出信号直接作用在被控设备之上，替代模拟量输出模块的 CV 输出量。

-Software Manual：软件手动控制方式，此时 PID 计算结果无效，在调试页面的 Set Output 项直接设置输出值百分比（即自动模式时的显示值），也可以通过例程梯级逻辑或人机界面直接设置输出控制量至 PID 结构数据标签子元素 PID1. SO，设置量将通过模拟量模块的输出通道 D/A 转换后的模拟量信号送至被控设备。

由此可见，硬件手动控制方式的确能覆盖软件手动控制方式。

- Tuning Constants：调试参数，PID 的主要调试参数用于调节 PID 的控制性能，不同的 PID 控制数学模型、独立增益模式 Independent 和相关增益模式 Dependent 的比例系数、微分系数和积分系数的物理意义是不相同的。

-Proportional Gain［Kp］：比例调节参数，独立增益模式对应的是无量纲的比例增益；
Proportional Gain［Kc］：比例调节参数，相关增益模式对应的是控制器增益。
比例调节参数的增益可调节输出控制变量的幅度，此值如果设置过大，即使没有设定微分调节参数，也有可能令调节对象发生超调现象，令系统发生振荡。

-Integral Gain［Ki］：积分调节参数，独立增益模式对应的是积分增益（1/秒）；
Reset Time［Ti］：积分调节参数，相关增益模式对应的是复位时间（分钟/每次循环）。
积分调节参数可调节控制精度，让控制变量的调节逼近静态工作点所需要的过渡过程时间，不同的运算模式为倒数关系，独立增益模式下，数值越小，过渡过程时间越长；相关增益模式下，数值越大，过渡过程时间越长。当过渡过程时间太长时，可能导致以为没有积分作用的误解。

-Derivative Time［Kd］：微分调节参数，独立增益模式对应的是微分增益（秒）；
Der ivative Rate［Td］:微分调节参数，相关增益模式对应的是加速时间（分钟）。
微分调节参数，在顺序控制系统，是控制速度要求不高的需求，比较少用到微分调节，大多设为 0 以不要微分作用。

-点击 Reset ，将复位调试参数。

以上 PID 性能参数可用手动调节，亦可使用 AutoTuning 软件自动调试。注意，大滞后系统不适合使用 AutoTuning 软件自动调试。

PID 指令的监视界面

每一个组态页面下端，都带有监视界面，以便随时查看 PID 的运行状态，如图 12-21 所示。

Setpoint (SP):	700.0		PV Alarm:	None
Process Variable:	0.0		Deviation Alarm:	None
Error:	0.0		Output Limiting:	None
Output:	0.0	%	Error Within Deadband:	No
Tieback:	0.0	%	Setpoint Out of Range:	No
Mode:	Auto		PID Initialized:	No

OK　Cancel　Apply　Help

图 12-21　PID 指令监视界面

监视界面参数说明：

- Setpoint（SP）：显示当前设定值。
- Process Variable：显示当前过程变量 PV。
- Error：显示误差极性，PV—SP 为正向；SP—PV 为负向。
- Output：显示当前输出百分比。
- Tieback：显示当前手动变量。
- Mode：指出自动、手动或软件手动方式。

- PV Alarm：指出 PV 报警状态，如果高位报警置 1，显示 HIGH；如果低位报警置 1，显示 LOW；如果没有报警，显示 None。
- Deviation Alarm：指出偏移报警状态，如果高位报警置 1，显示 HIGH；如果低位报警置 1，显示 LOW；如果没有报警，显示 None。
- Output Limiting：指出输出超限状态，高超限显示 HIGH；低超限显示 LOW；没超限显示 None。
- Error Within Deadband：误差在死区范围显示 Yes，否则显示 No。
- Setpoint Out of Range：设定值超出范围显示 Yes，否则显示 No。
- PID Initialized：如果定标页面此项被选择，则显示。

第 13 章

通信指令 MSG 的编程

MSG 指令是 Logix 控制器对外部设备实行通信操作的指令，这些外部设备可以是模块，也可以是单一的设备，只要具有通信能力的设备，控制器几乎都可以通过 MSG 指令与之实施信息交换。这条指令功能很强，不但能够在 Logix 系列的控制器之间完成批量数据的传送，还能跟传统产品 PLC5/SLC500 处理器交换数据，此外还可以对模块或设备发送服务性指令，实施操作。当控制器执行 MSG 指令时，作为信息发动者，可以对目标设备读数据或写数据，实现了数据的双向传送。

13.1　分配给 MSG 指令的结构数据

在控制器数据区域中创建一个 MESSAGE 类型结构数据标签，分配给一条 MSG 指令，也可以在 MSG 指令的 Message Control 项直接键入结构数据标签名称，在该项处右击后，点击 New Tag…，进入创建新标签的界面，这时系统会自动选定 MESSAGE 类型的结构数据。总之，每条 MSG 指令都必须对应一个 MESSAGE 类型结构数据标签，用来记录本条指令的组态信息和执行状况。

MSG 指令是系统对外操作的指令，MESSAGE 类型结构数据包含了本条指令的组态信息和状态信息，由于指令执行的状态就是对外部设备通信操作的当前状态，所以像 I/O 模块组态一样，这个结构数据标签必须放在控制器区域，即外部数据区域，且不能用作为结构数据数组标签，只能创建为单一的结构数据标签。

当梯形图例程的梯级逻辑使能，MSG 指令被启动后，不能当即执行完毕，而是进入控制器的通信缓冲区排队，等待通信机会，一旦获得，则跟设备通信，进行数据的交换，直到数据交换完毕，指令执行才会完成。在此过程中，指令将处于不同的工作状态。如图 13-1 所示是 MESSAGE 类型结构数据中的状态位和设置位部分。

Msg_CML
Msg_CML.Flags
Msg_CML.EW
Msg_CML.ER
Msg_CML.DN
Msg_CML.ST
Msg_CML.EN
Msg_CML.TO
Msg_CML.EN_CC

图 13-1　MESSAGE 类型结构数据中的状态位和设置位

指令状态位：

- EW（等待位）：当 MSG 指令请求进入通信缓冲区排队，该位置位，指令处于等待状态；直到开始位（ST）置位时，等待状态结束，EW 复位。
- ER（错误位）：MSG 指令数据传送失败时，

该位置位，MSG 指令梯级条件重新触发时自动复位该位。

- DN（完成位）：当 MSG 指令的最后一个数据包成功传送完毕，该位置位，MSG 指令的梯级条件重新触发时自动复位该位。
- ST（开始位）：MSG 指令的数据开始传送，该位置位，直到数据传送结束，不管是否传送成功，当错误位或完成位置位时自动复位该位。
- EN（使能位）：MSG 指令梯级条件触发，指令被使能，该位置位并保持，即使梯级条件变假，也仍然保持，直到 DN 或 ER 置位才复位；如果梯级条件变假，但 DN 或 ER 被清除，处于复位状态，则仍然保持 EN 位置位。总之，MSG 指令使能位与梯级条件并不一致，还跟 DN 位和 ER 位的状态有关。

以上 5 个位为只读数据，不可更改，是系统给出的 MSG 指令执行时的状态。此为结构数据标签的子元素，可以送至人机界面用于提示，如错误位置位的状态。

指令设置位：

- TO：可在 MESSAGE 结构数据标签中手动直接设置该位，如果设定，控制器停止处理数据传送，并置位 ER，MSG 指令使能时将自动清除该位。
- EN_CC（使能连接位）：用于管理 MSG 指令的缓存连接，这个位的设置状态取决于 MSG 指令通信页面的 ☑ Cache Connections 选项，当此项被选定，该位为 1；当此项被取消，该位为 0。同样，可编写梯级逻辑实现 EN_CC 位的变更，从而动态地管理 MSG 指令的缓存连接。

以上两个位是设置数据，可以在数据表中的结构数据标签中直接设置，也可以通过编写梯级逻辑来动态设置。

为了更清楚地了解各状态位之间的依存关系，作为正确运用状态位的依据，让编写的 MSG 指令能够精确无误地执行，下面我们通过如图 13-2 所示的状态过程图来讨论 MSG 指令执行过程中状态位的变化。

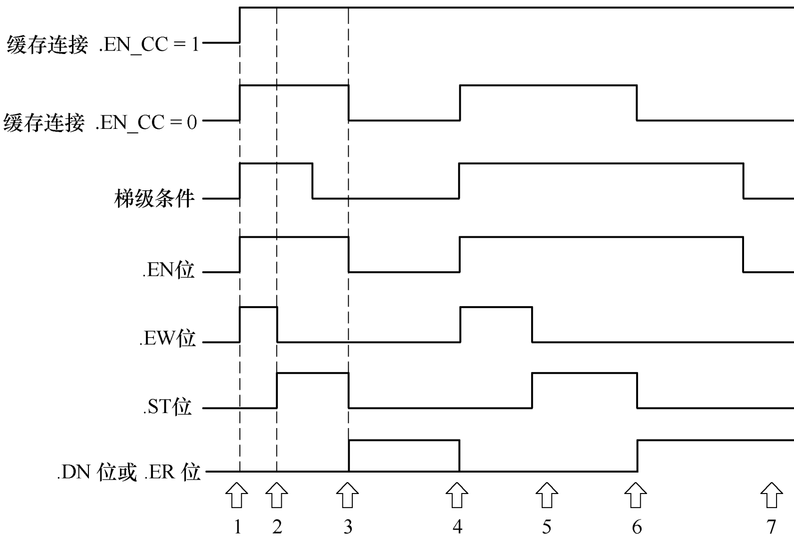


图 13-2　状态过程图

状态位变化 7 步过程讨论：

- 第 1 步：梯级条件变为真，EN 置位，EW 置位，缓存连接打开，EN_CC 位设置为 1，缓存连接保持打开；EN_CC 位设置为 0，缓存连接启动打开。
- 第 2 步：数据开始传送，ST 置位，EW 复位，缓存连接仍然被打开。期间梯级条件变为假，EN 保持置位。
- 第 3 步：MSG 指令数据传送完成或出错，梯级条件已经变为假，使能位复位，ST 复位，DN 或 ER 置位。EN_CC 位设置为 1，缓存连接保持打开；EN_CC 位设置为 0，缓存连接关闭。
- 第 4 步：梯级条件再变为真，EN 置位，EW 置位，缓存连接打开，EN_CC 位设置为 1，缓存连接保持打开；EN_CC 位设置为 0，缓存连接启动打开。先前 DN 或 ER 的置位状态被复位。
- 第 5 步：数据开始传送，ST 置位，EW 复位，缓存连接仍然被打开。期间梯级条件一直保持为真，EN 保持置位。
- 第 6 步：MSG 指令数据传送完成或出错，梯级条件仍然为真，EN 仍然置位，ST 复位，DN 或 ER 置位。EN_CC 位设置为 1，缓存连接保持打开；EN_CC 位设置为 0，缓存连接关闭。
- 第 7 步：梯级条件变为假，使能位被复位，DN 或 ER 仍保持置位。

可见，MSG 指令一旦被触发，不管梯级条件是否持续存在，指令都处在使能状态，直到工作结束为止。EW、ST 以及 DN 和 ER 依次置位，而且 DN 和 ER 的置位状态要保持到下一次梯级条件触发才会清除。

缓存连接如果设定为固定连接，即 EN_CC 位设置为 1，则一直都保持连接，不会释放；缓存连接如果设定为动态连接，即 EN_CC 位设置为 0，在指令等待和数据传送期间保持缓存连接，其余时间则释放缓存连接，只有等待和交换数据时才会占用连接。

如果 MSG 指令 ER 置位，错误代码和扩展错误代码将存放在 MESSAGE 结构数据标签的子元素中，如图 13-3 所示。尽管错误代码在指令的监视页面同样可以读到，但作为结构数据标签的子元素是可以送至人机界面作为提示信息的。

+ Msg_CML.ERR	16#0000
+ Msg_CML.EXERR	16#0000_0000

图 13-3　错误和扩展错误代码存放

13.2　MSG 指令通信的编程

根据以上的讨论，我们来开始编写 MSG 指令。如果希望一条 MSG 指令不停地反复执行，梯级条件则使用本条指令 MESSAGE 结构数据标签的使能位 Msg_CML.EN 的常开输入指令，如图 13-4 所示。

要让 MSG 指令按照我们的要求传送数据或执行某些操作，必须对指令进行组态。下面我们来讨论如何对 MSG 指令进行组态。双击 MSG 指令中的 ... ，打开如图 13-5 所示的组态页面。页面默认是 CIP Generic 的组态，这是服务性指令的操作，根据相应的资料提供，对



图 13-4 连续执行 MSG 指令的梯级逻辑

服务类别和指定的服务代码等参数进行设定。

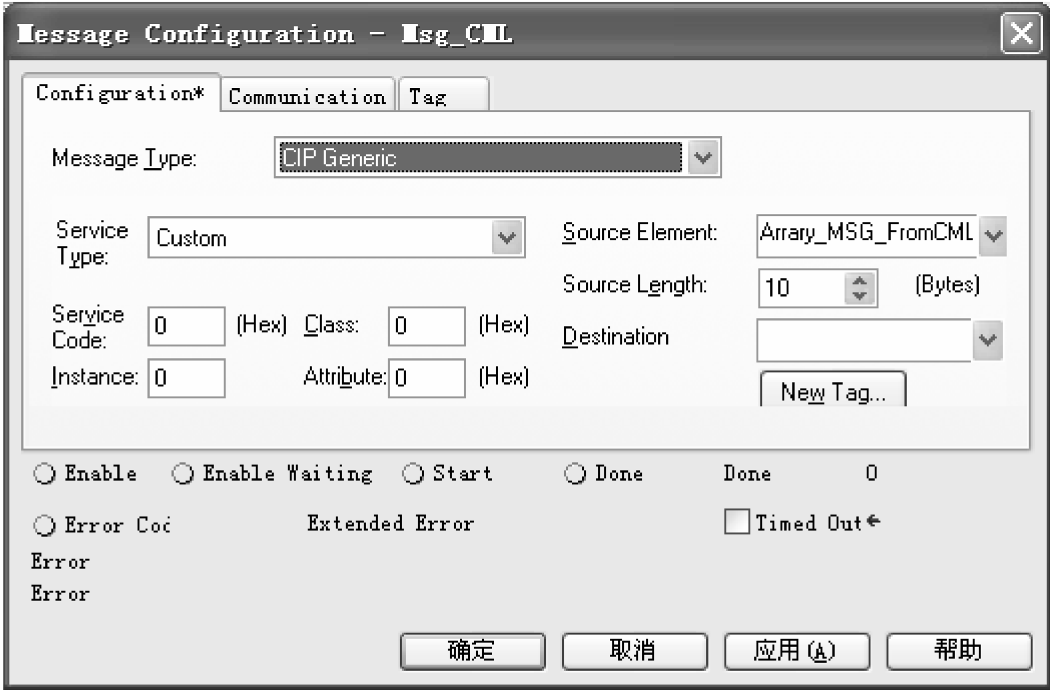


图 13-5 MSG 指令组态页面

我们先讨论 Logix 控制器之间数据交换的通信，打开 Massage Type，如图 13-6 所示，这里是通信类型的选择，关于其他数据通信的类型，我们不作详细讨论。其中，CIP Data Table Read 和 CIP Data Table Write 两种通信类型是最常用的，用来完成 Logix 控制器之间的读写通信操作，我们以此为例来讨论。

选择通信类型 CIP Data Table Read，如图 13-7 所示，是关于本控制器向对方控制器读取数据的通信操作。源数据元素为对方控制器数据标签地址，需要键入对方控制器数据标签地址，目标数据元素为本控制器数据标签地址，可以在本控制器数据区域中已有的数据标签中指定，亦可临时创建。

选择通信类型 CIP Data Table Write，如图 13-8 所示，是关于本控制器向对方控制器写入数据的通信操作。目标数据元素为对方控制器数据标签地址，需要键入对方控制器数据标签地址，源数据元素为本控制器数据标签地址，可以在本控制器数据区域中已有的数据标签中指定，亦可临时创建。

由此看来，不同的数据传送方向，读数据或写数据决定了 MSG 指令发动控制器的源数据地址和目标数据地址不同，本控制器的数据标签可以在数据库中选定，被访问控制器的数

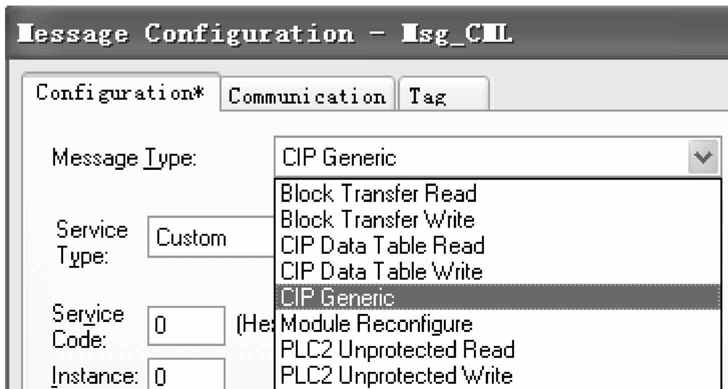


图 13-6　通信类型的选择

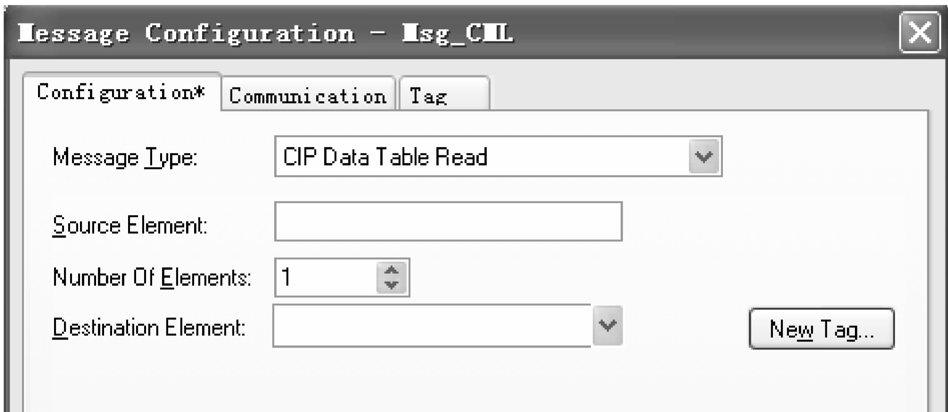


图 13-7　选择通信类型 CIP Data Table Read

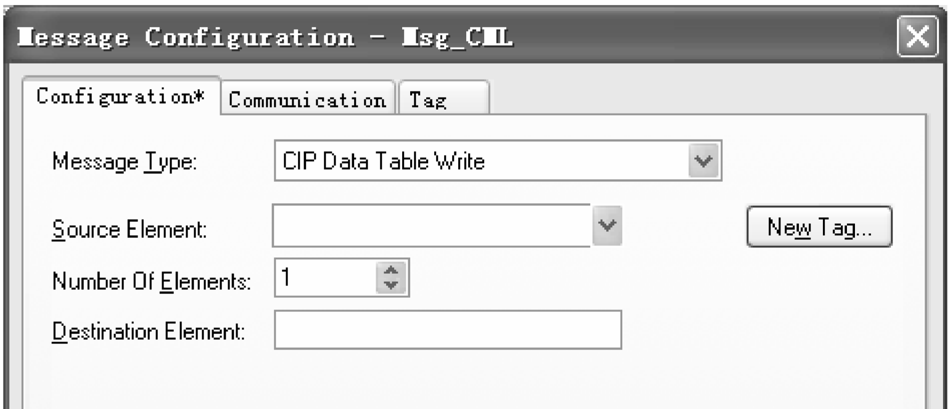


图 13-8　选择通信类型 CIP Data Table Write

据标签则要键入，有时描述性标签的构成不是特别简单的书写，建议从被访控制器的项目文件的数据库中复制过来，以免键入的微小错误导致 MSG 指令执行时访问失败，带来调试的麻烦。

临时创建通信数组标签，点击 New Tag... ，进入创建页面，如图 13-9 所示。注意，作为对外通信的数据标签，该标签要创建在控制器数据区域，且为 32 位的结构数据标签或数组标签。可以注意到，在 MSG 组态页面选择本控制器要传送的数据标签时，可供选择的数据区域只有控制器数据区域，程序数据库区域是灰色的不可选择的。

数组标签创建的元素个数应该大于等于 MSG 指令要传送的数据元素个数，否则 MSG 指令执行时，将由于操作数组元素不足而报错。

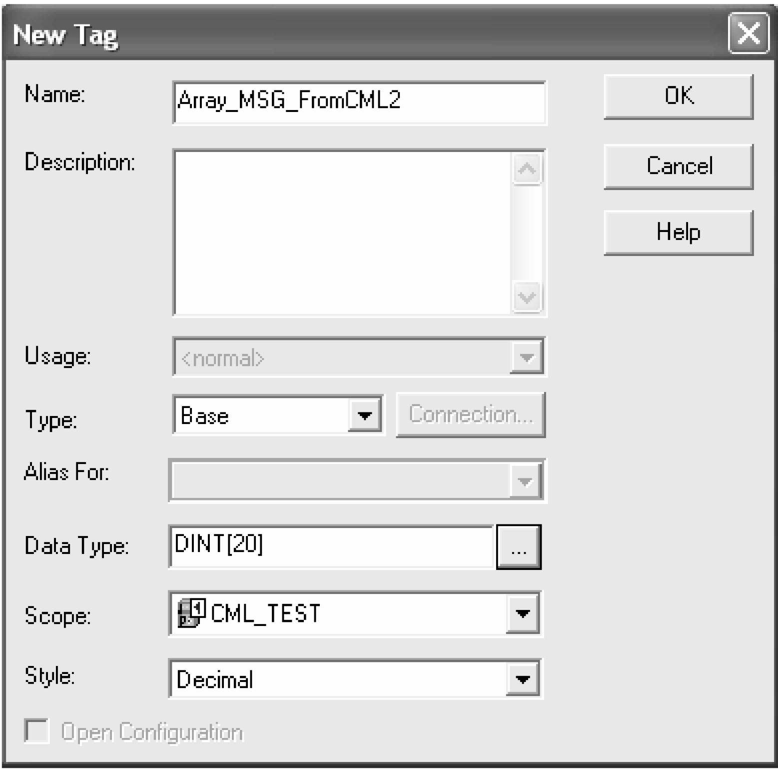


图 13-9　新标签创建页面

组态完成的页面如图 13-10 所示。要特别注意的是，选择源数据标签地址和目标数据标签地址时，必须是要传送的数组标签指定的首个元素，而不是单一的数据标签名称，然后在元素数目的栏目 Number Of Element 中键入要传送的元素个数。这样，首址加要传送的元素个数概括了数组中被传送数据的范围。如果不小心写成了单一的数组标签名称而没有指定首个元素编号，在较高控制器版本的系统中将默认为首个元素是 0 号元素，较低控制器版本的处理则可能是只传送第一个元素，后面的元素不予传送，一定要留意这一点。

刚才的组态页面只指定了相互通信的两个控制器的源数据标签地址和目标数据标签地址，并没有指定信息通信的路径。路径有一定的结构规则，由一个或多个路段构成，路段复杂时可以通由不止一个网络，甚至可以是不同类型的网络，这也是 MSG 指令传送灵活的特点之一。

MSG 指令通信组态要填写通信的路径，路径的书写规则是：

- 一条路径由多个路段组成，每个路段的表达是X，Y。通常一条从本控制器出发到

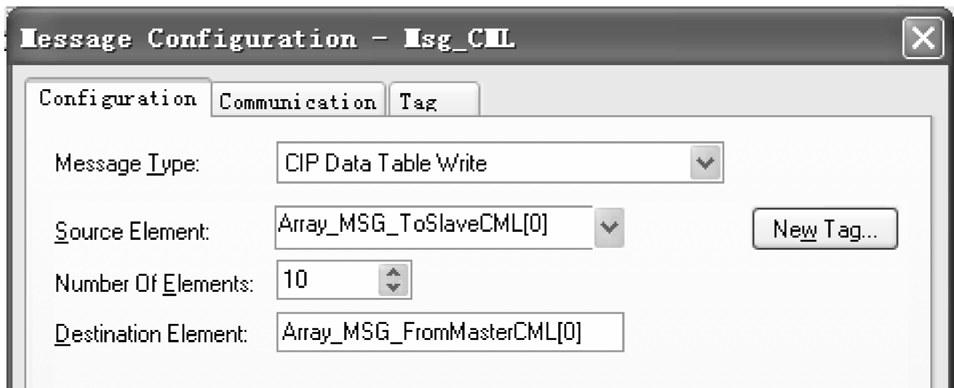


图 13-10　组态完成的组态页面

达对方控制器的路径会表达为

X，Y，X，Y，X，Y

- X　表示背板或网络，背板为 1，网络为 2。
- Y　表示槽号或站号，槽号在 0 ~ 16 之间，站号则不同网络的表达形式不同：

  ControlNet　　1 ~ 99 十进制

  DH +　　00 ~ 77 八进制

  EtherNet/IP　　IP 地址

点击进入通信组态页面 Communication，如图 13-11 所示。这里指定的通信路径是从一个 CompactLogix 控制器出发，到达一个 IP 地址为 192. 168. 1. 111 的另外一个 CompactLogix 控制器所经历的路径，两个控制器都位于背板总线的 0 槽。信息出发的控制器即为 MSG 指令发动的控制器，信息路径与数据传送方向无关，不论 MSG 指令的操作是读信息或是写信息，路径总是从 MSG 指令发动的控制器指向被 MSG 指令访问的控制器。

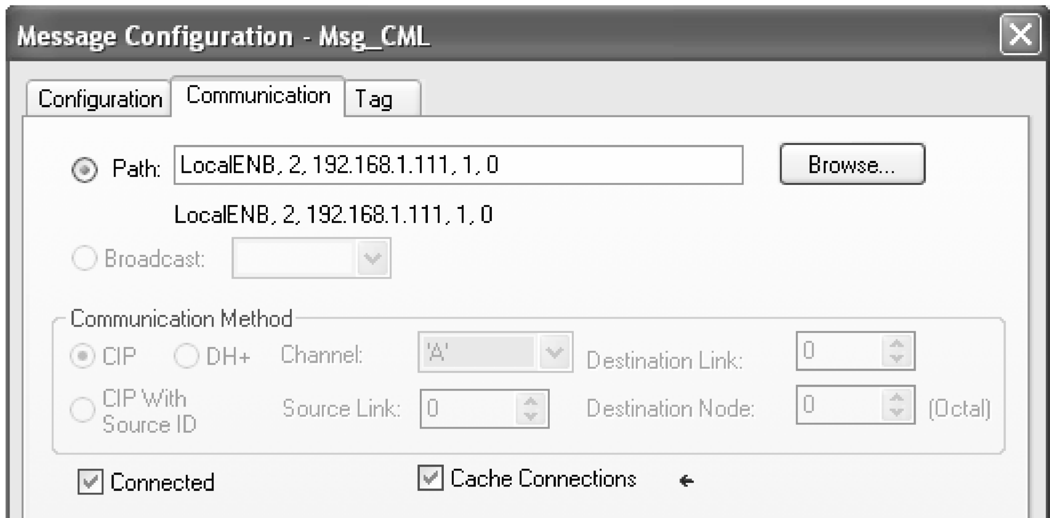


图 13-11　MSG 指令的通信组态页面

在通信组态页面，填写从 MSG 信息发动的控制器出发，直至到达对方控制器的所经过的路径。这种路段书写规则，不仅仅使用在 MSG 指令，在一些产品的访问路径上也都在使

用，但路段的隔离符号却略有差异。MSG 指令所使用的隔离符号是逗号，有的产品则使用空格或分号作为分隔符号。

Cache Connections 是缓存式连接，此项被勾选，表示这条 MSG 指令固定的占用一个控制器的连接；如果取消，表示只有在运行这条 MSG 指令时才会占用控制器的连接。前面讨论的 MSG 指令执行状态就包含了此项的设定。

缓存式的连接在控制器中都是有限量的，每个控制器不能超过 32 个，这意味着如果这种通信类型的 MSG 指令在同一个控制器中的使用要超过限量，就不得不取消 Cache Connections 的选项，编制轮流执行 MSG 指令的梯级逻辑，并互锁执行。这 32 个缓存式连接是包含在控制器总连接数中的，也就是说，MSG 指令每消耗一个缓存式连接的同时也消耗了控制器的一个连接。

另外，容易被误解的是，以为控制器被限于只有 32 条 MSG 指令可执行，实际上是被限于只有 32 个缓存式连接，除了 Logix 控制器之间 CIP 式的通信类型和块传送会要占用缓存式连接，服务性指令和与传统产品处理器通信的 MSG 指令并不占用缓存式连接。不是说非 CIP 式的通信类型便可节省控制器通信资源，它们只是不占用缓存式连接的资源，却要占用非连接缓冲区的连接，那里的资源更为紧张，在这里我们不作深入的讨论。

仅仅是从控制器通信资源受限的角度来看，当为数不少的 MSG 指令需要发送时，不得不认真考虑如何编写指令执行的梯级逻辑，否则将造成通信的障碍，令 MSG 指令不能成功执行。下面我们讨论如何通过梯级逻辑的控制来安排 MSG 指令的执行。

如果需要几条 MSG 指令互锁执行，只要按照如图 13-12 所示的梯级逻辑编写就可以了，注意梯级条件的输入位逻辑的顺序，将本条指令的 EN 常开位放在离指令最近的位置，其他指令 EN 常开位则按照本条指令之后的执行顺序排列在前面。这样的编写，可使得指令的执行顺序是：

Msg_CML1 ⟶Msg_CML2 ⟶Msg_CML3

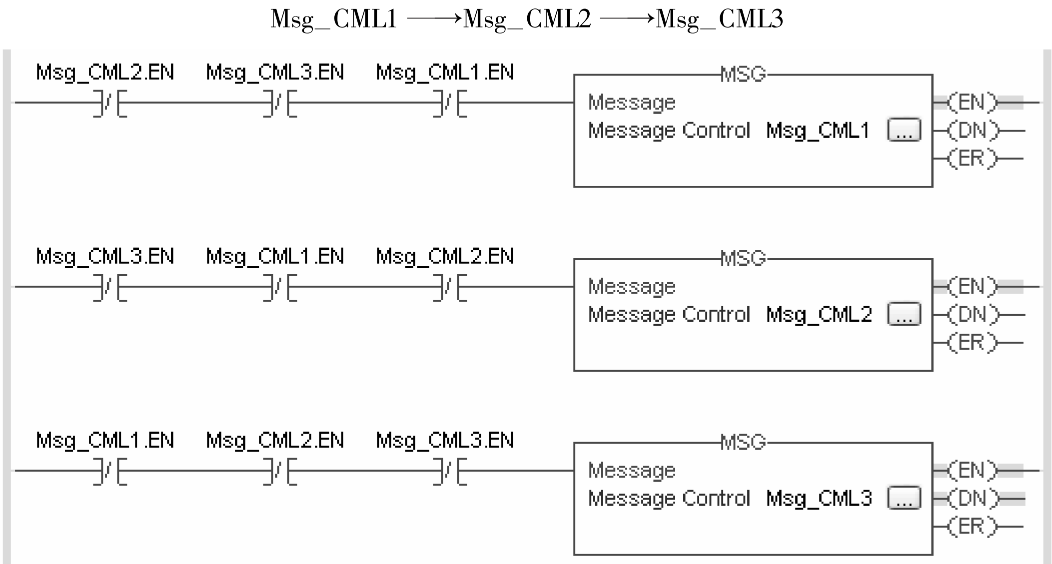


图 13-12　互锁按序的 MSG 指令执行

这种编程方法不仅仅适用于 MSG 指令，只要是希望指令依照顺序反复执行，大都采用这样的编程方式，在编程基础的章节中，我们曾经讨论过。这里是 MSG 指令执行的实例，

在 MSG 指令不多的情况下，这是通常的做法，可以保证 MSG 指令的顺畅执行。

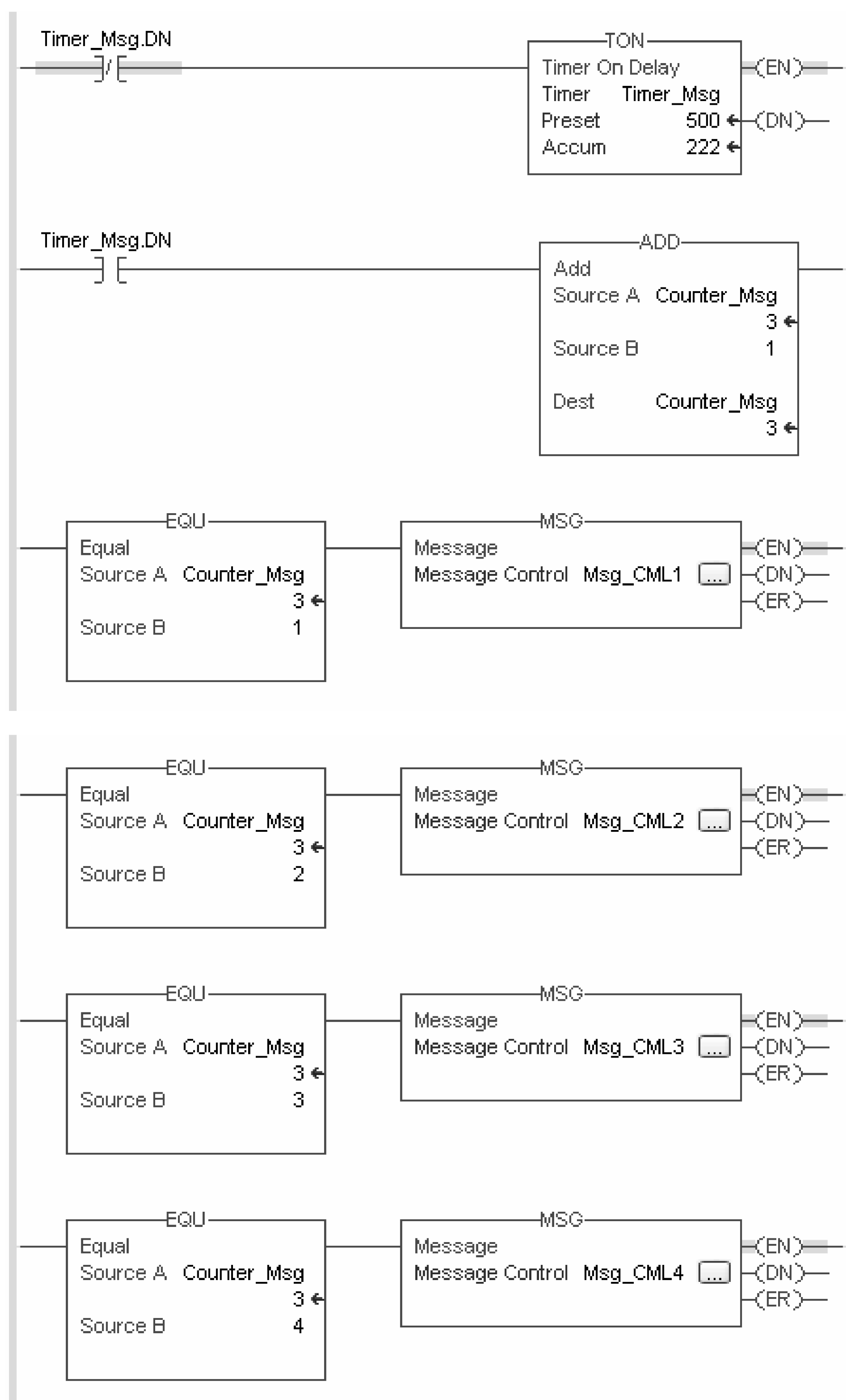


图 13-13　指针控制 MSG 指令执行的进程

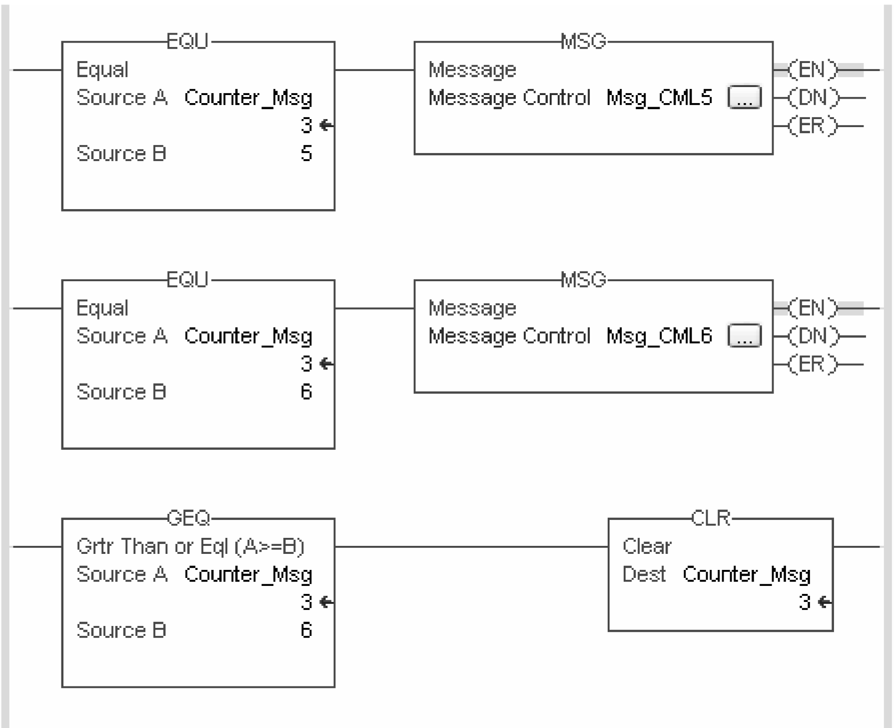


图 13-13　指针控制 MSG 指令执行的进程（续）

如果有更多的 MSG 指令需要互锁执行，梯级逻辑的输入指令部分则显得繁杂，不妨采用指针的方法来控制进程，如图 13-13 所示是常用的一种梯级逻辑编写实例。

设定 0.5s 传送一条 MSG 指令，如果网络正常，足以保证 MSG 指令能传送成功，所有数据传送一遍的周期需要 3s 时间。这样的编程适合数据交换周期时间长，没有苛刻的时间限制，一旦某条 MSG 指令传送不成功，不会影响到下一条 MSG 指令的传送。如果通信页面设置的 Cache Connections 项是被取消的，这么多的 MSG 指令传送只会占用 1 个缓存连接。此外，每条 MSG 指令梯级条件的维持时间是 0.5s，要确认 MSG 指令是梯级条件跳变才会执行的，所以这样的梯级条件不会令 MSG 指令不停地被触发。

如果为了紧凑地使用通信时间，高效率地传送数据，让 MSG 指令一条接一条的执行，又要保证前一条确定执行完毕才执行下一条 MSG 指令，则要利用 MSG 指令的状态位来控制进程了。梯级逻辑的编写如图 13-14 所示。

这种编程方式，前一条执行完成位置位才可以令下一条 MSG 指令的梯级条件成立，其风险是，一旦其中一条 MSG 指令执行有问题，那么所有的 MSG 指令都没法继续执行。这种应用只适合的场合为：任一条 MSG 指令没有送出，其他 MSG 指令的传送将变得没有意义。

第一条 MSG 指令用初始位 S: FS 触发，也可以用复位位 Reset_MSG 来触发，之后便进入 6 条指令轮流执行的循环之中。复位的操作时间不能超过 6 条 MSG 指令执行的循环时间，否则会产生执行顺序问题。这在人工操作中是难以控制的，所以要用 ONS 指令来确保短脉冲操作动作。

前一条 MSG 指令执行后的完成位将作为本条 MSG 指令的梯级条件，执行本条 MSG 指令并复位梯级条件的完成位，然后进入下一条 MSG 指令的执行。

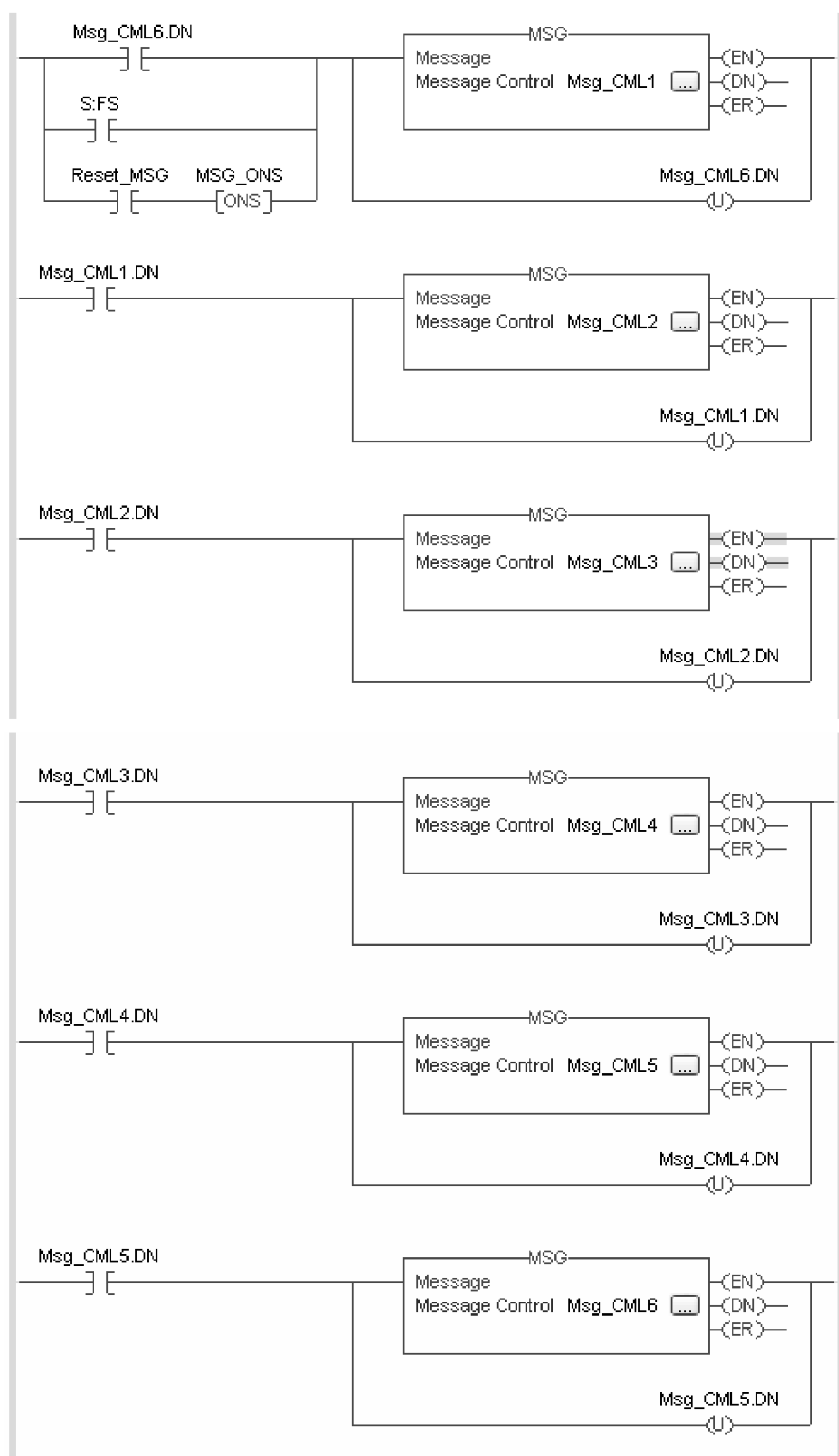


图 13-14　MSG 指令的连续执行

考虑到某条 MSG 指令发生错误而引起的卡壳现象，即使错误已经消除，但是曾发生错误的 MSG 指令已经失去了梯级条件触发复位 ER 的机会，使得后面的 MSG 指令不能继续执行。不妨编写一条梯级逻辑用来探测各条 MSG 指令的错误，并设置提示位 MSG_Error，当看到错误提示位置位后，可操作复位 Reset_MSG 来恢复正常的运行，如图 13-15 所示。不要试图用提示位 MSG_Error 作为并行的梯级条件来直接复位，这可能给你带来意想不到的麻烦。

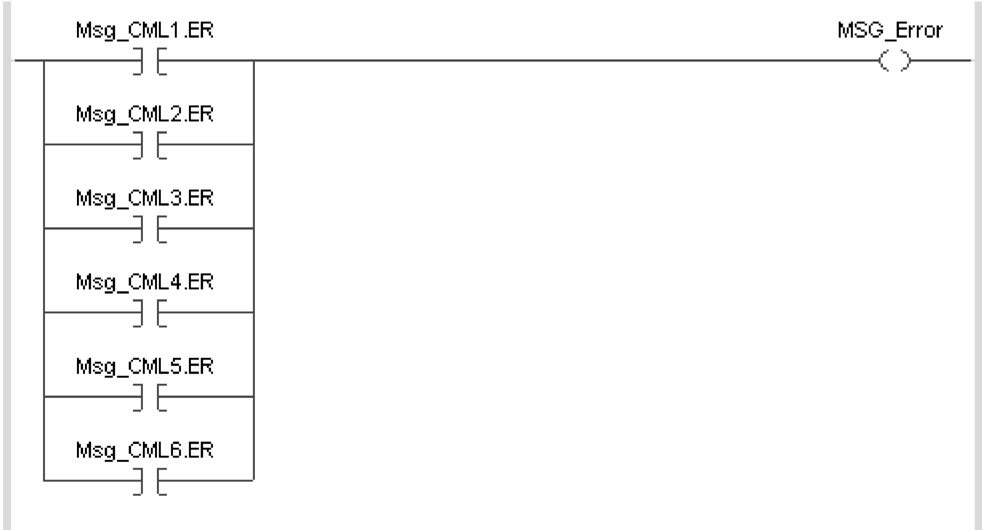


图 13-15　MSG 指令错误提示

从以上 3 个实例讨论可以看出，如何安排 MSG 指令的执行顺序完全取决于通信需求，在满足需求的前提下，尽可能地减轻控制器的通信负担，节省控制器通信资源。

有时会遇到在同一个控制系统内主控制器向所有从控制器发布相同信息的应用，那么我们可以巧妙地运用修改通信路径的方法来实现。MESSAGE 结构数据中的一个子元素是用来存放通信路径的，如图 13-16 所示。通过编写逻辑代码执行替换这个子元素，也就替换了 MSG 指令的通信路径。

Name	Value
+ Msg_Master_CML.Path	'$01$01$12$r192.168.1.101$00$01$00'

图 13-16　MSG 指令通信路径子元素

首先创建一个用户自定义的字符串数据类型，长度为 30 个字符，比可能用到的路径字符串要更大一些，用来存放书写通信路径的 ASCII 码数据。对于同一个网络的 MSG 信息路径描述，30 个字符基本就足够了，如图 13-17 所示。MESSAGE 结构数据标签路径子元素的字符串长度是 82 个字符的 STRING 类型，我们没有必要定义这么长的字符串数据类型，满足需求就好，尽可能节省内存空间。

在数据区创建 6 个元素的数组标签，选择数据类型为刚刚创建的 STRING_MSG_PATH，并在数据输入的字符串面板中键入将要使用的路径描述。如果对键入的字符串数据没有把握，可以从通过组态页面已经产生的路径子元素复制而来，然后局部修改。如图 13-18 所示是在数据表中已经创建并键入的字符串数据的数组标签，它将用于主控制器对 6 个从控制器

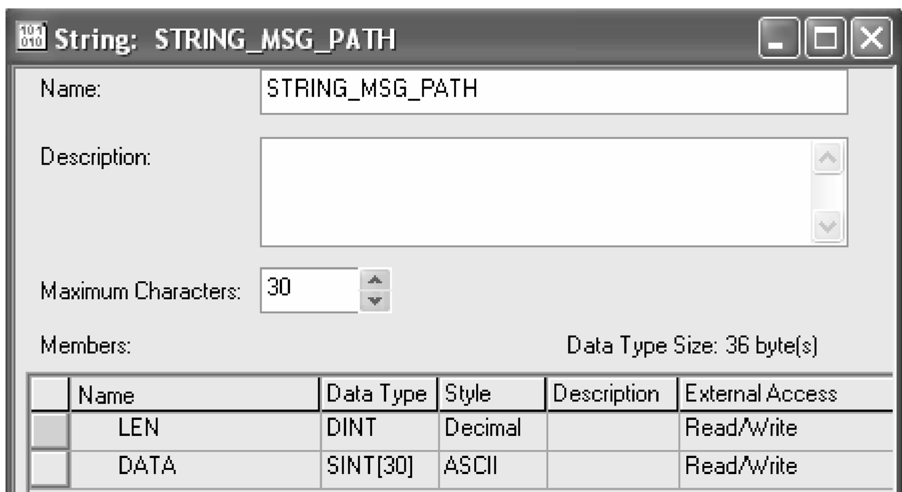


图 13-17　定义 MSG 通信路径的结构数据

访问的路径描述。

Name	Value
Msg_Master_Path_Buffer	{...}
Msg_Master_Path_Buffer[0]	'$01$01$12$r192.168.1.101$00$01$00'
Msg_Master_Path_Buffer[1]	'$01$01$12$r192.168.1.102$00$01$00'
Msg_Master_Path_Buffer[2]	'$01$01$12$r192.168.1.103$00$01$00'
Msg_Master_Path_Buffer[3]	'$01$01$12$r192.168.1.104$00$01$00'
Msg_Master_Path_Buffer[4]	'$01$01$12$r192.168.1.105$00$01$00'
Msg_Master_Path_Buffer[5]	'$01$01$12$r192.168.1.106$00$01$00'

图 13-18　创建有关的通信路径标签

编写的梯级逻辑如图 13-19 所示，仍然采用指针控制的方法，根据不同的指针判定，确定复制某个新的路径给 MSG 指令结构数据标签的路径子元素。此例将指向 6 个不同 IP 地址的从控制器，用累加计数修正指针，轮流对从控制器传送数据，每逢 6 便复位指针，重新开始下一轮数据传送。

建立一个自复位的定时器，每 0. 3s 产生一个 DN，作为指针增量操作动作，并同步传送一次数据。对所有的从控制器通信共同使用一条 MSG 指令，每当指针增加一次，MSG 指令便执行一次，将指定的数据内容送往指令通信路径所指向的从控制器。

勾选 MSG 指令通信页面的 Cache Connections 项，让 MSG 指令的通信固定地占用一个缓存连接。

请留意 COP 指令的长度，这里是复制路径字符串的数据的个数，从第一个开始，直到描述结束，这个长度可以自己算出，简单的办法是直接到数据库去读取一个指令当前的字符串数据，如图 13-20 所示。用这个方式直接且结果正确，不必纠结中间字符是否算数。

其实，稍稍研究一下 COP 指令的源地址和目标地址会觉得有点问题，如果目标地址是 Msg_Master_CML. Path，这是 MESSAGE 结构数据标签的一个子元素，源地址 Msg_Master_Path_ Buffer是数组标签的一个元素，COP指令的长度应当为1才对，是一个元素的复制操

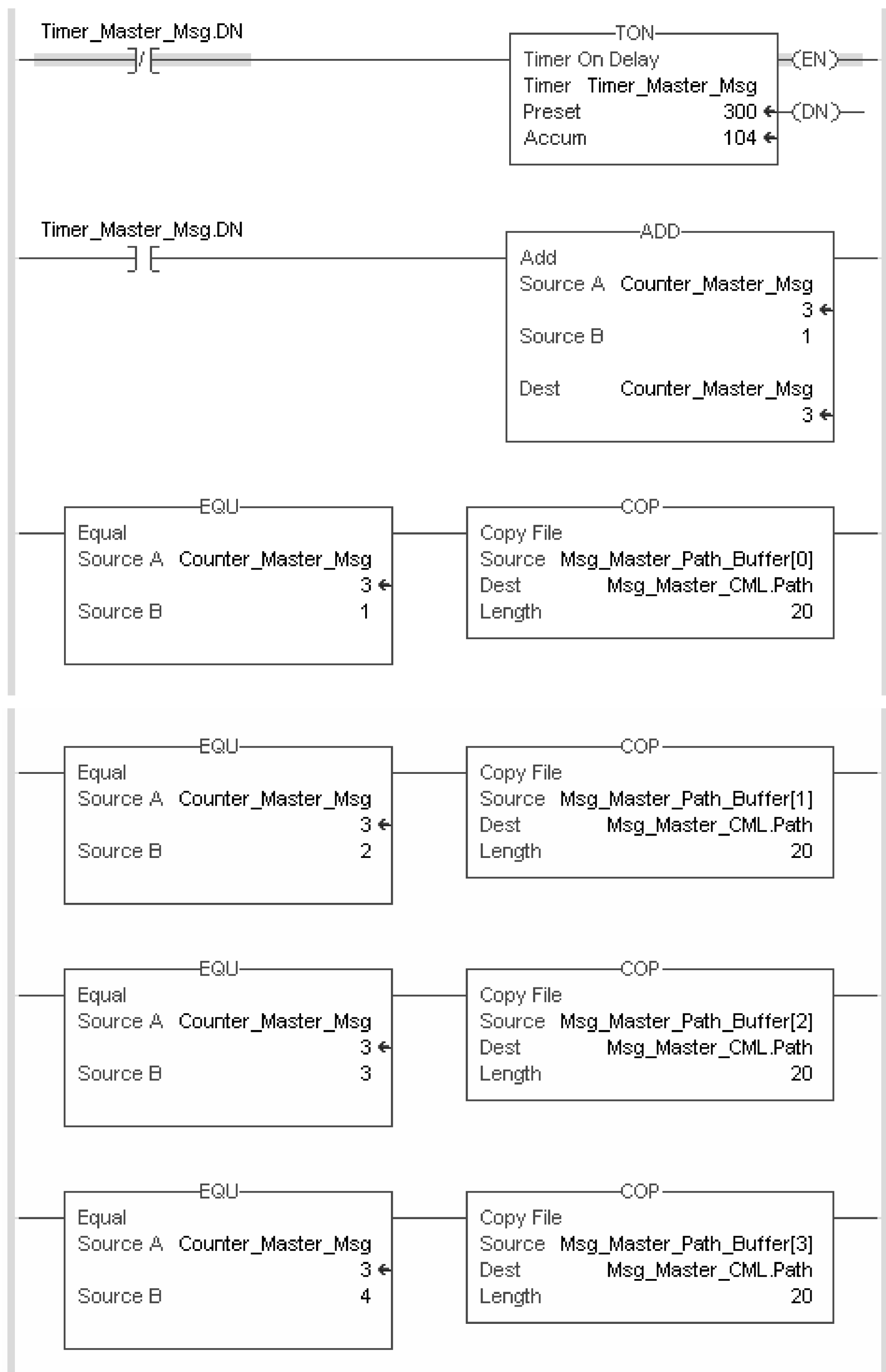


图 13-19　修改通信路径对不同的从控制器发送 MSG

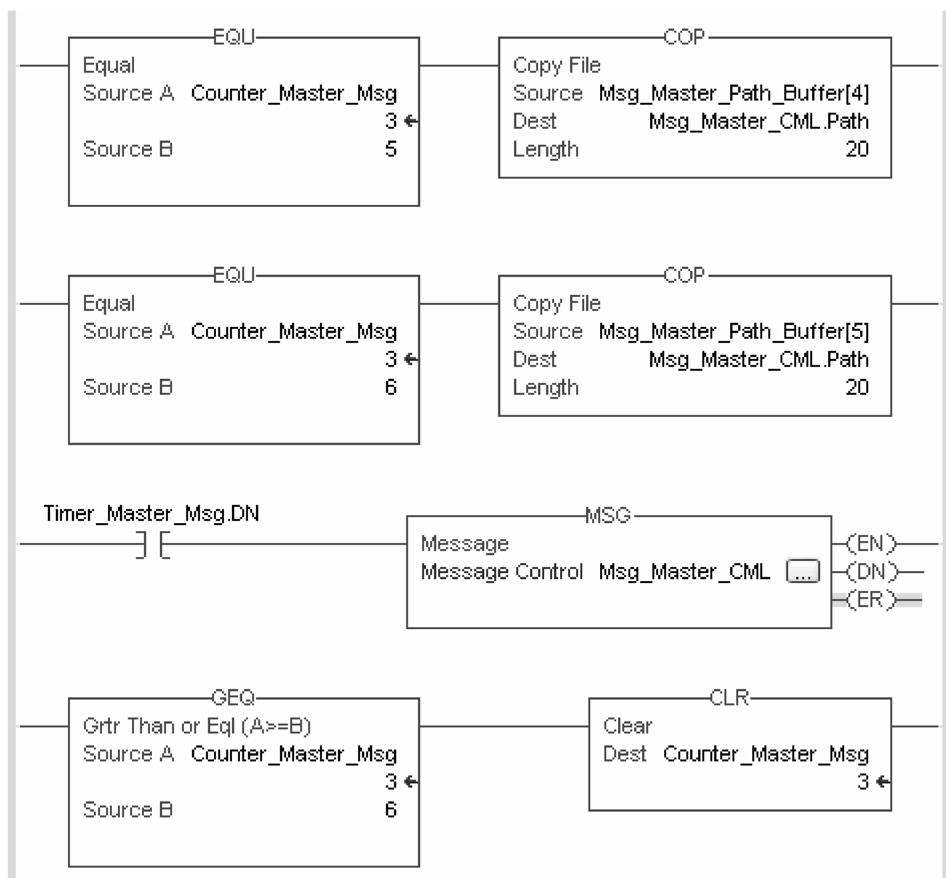


图 13-19　修改通信路径对不同的从控制器发送 MSG（续）

− Msg_Master_CML.Path	'$01$01$12$r192.168.1.101$00$01$00'
+ Msg_Master_CML.Path.LEN	20
+ Msg_Master_CML.Path.DATA	{...}

图 13-20　获取路径子元素的长度

作，但这两个元素的字符长度是不一致的。这样看来似乎目标地址用 Msg_Master_CML. Path. DATA［0］才对，COP 指令的长度就是字符个数，如此一来，源地址又怎么表达它的首址呢？只好测试一下了，结果如图 13-19 所示的那样，直接将 Msg_Master_CML. Path 作为目标地址，在 Msg_Master_CML. Path 中可观察到目标地址的变化是正确的。

可惜最终的 MSG 指令执行没法测试，6 个从控制器的通信路径访问只能到现场去印证了，好在我们有了一条 MSG 指令访问一个控制器的测试经验，即使现在 MSG 指令给出 ER 错误提示，我们仍然能预测到正确的结果。这个实验只是帮助我们了解通信路径是否能动态修改，从而编写简单的 MSG 指令。

MSG 指令的组态页面设置如图 13-21 所示。在接收主控制器信息的每个从控制器中，最好创建名称和结构都相同的数组标签，比如标签名为 From_MasterController 的主控制器目标地址的数据标签。

在从控制器一方创建名称和结构都相同的数组标签，为组态页面提供一个固定的地址自

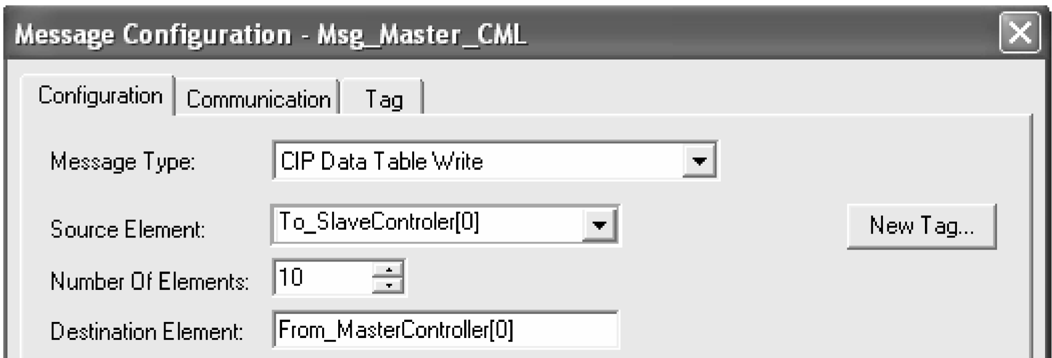


图 13-21　MSG 指令的组态页面设置

然是方便简洁，但是如果不得不修改目标地址，也是可以做到的，控制结构数据标签中还有一个子元素是描述对方控制器数据标签地址的，如图 13-22 所示。



图 13-22　描述对方控制器数据标签地址的子元素

这样，在同一个梯级中就必须同时有两个子元素 COP，如图 13-23 所示。

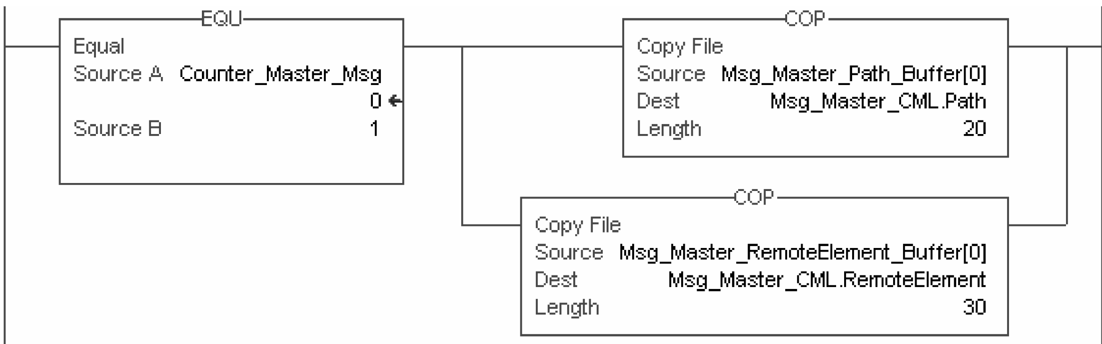


图 13-23　路径和对方控制器标签的复制

当然，跟创建通信路径标签数组的做法一样，在数据库中事先也要创建相应的标签数组 Msg_Master_RemoteElement_Buffer，并设置对方控制器数据标签地址的字符串数据。

13.3　MSG 指令的服务性操作编程

以上讨论的都是 Logix 控制器之间数据传送的 MSG 指令的运用，MSG 指令用途很广，常见的还有服务性指令操作。下面我们再来讨论 MSG 指令的另外一种运用。

针对控制器所在的系统中任何一个具有通信能力的设备或模块，都可以通过 MSG 指令来执行服务性指令操作，如图 13-24 所示。正如我们一贯的做法，为确保指令只执行一次，采用 ONS 指令限定梯级条件为短脉冲。

MSG 指令组态页面如图 13-25 所示，这是一个对设备复位的操作。一旦选中服务操作内容，服务代码等参数自动显示在下面。如果某个设备需要控制器通过 MSG 指令来发送某个服务性指令，在 CIP Generic 信息类型下选择 Custom 服务类型执行，并键入由设备厂家的技

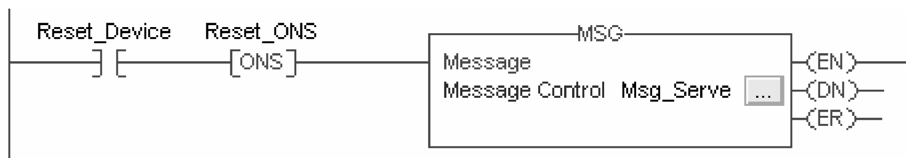


图 13-24　确保 MSG 指令唯一操作的梯级逻辑

术手册提供的相应的服务代码等参数。

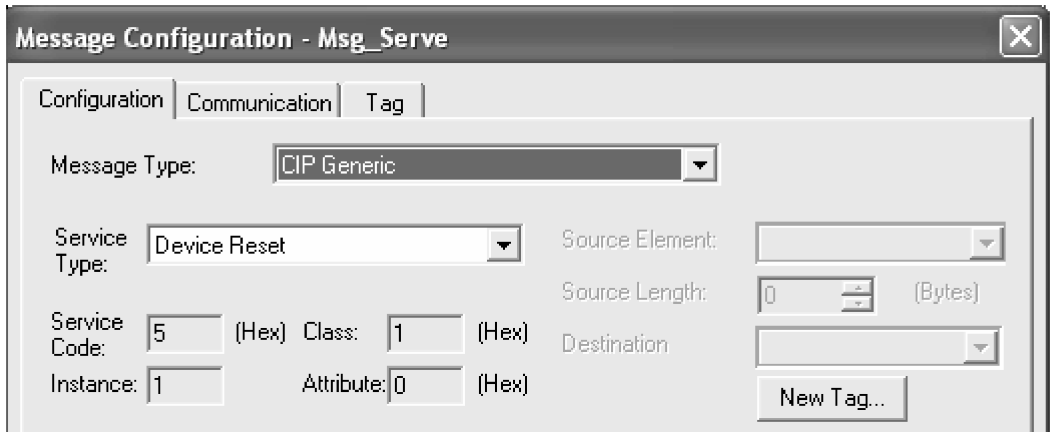


图 13-25　对设备复位的服务性指令操作

查看一下通信组态页面，如图 13-26 所示，通信路径指向了被操作的模块或设备。同时，可以在有关连接的设置中看到，这样的指令是不会占用缓存连接的。

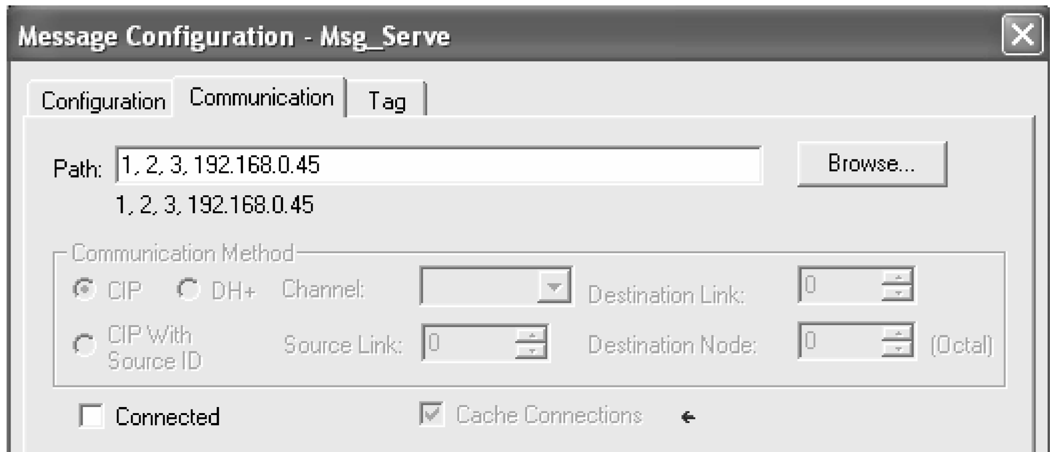


图 13-26　通信组态页面

不占用缓存连接的操作，又被称为非连接性操作，通过非连接缓冲区进入或发送，这就是非连接缓冲区通信资源的运用。

在讨论 Logix 控制器通信资源时，有如图 13-27 所示的示意图，每个控制器的非连接缓冲区有 3 个接收排队（Incoming）和 10 个发送排队（Outgoing），这是通信交换缓冲区的限量，是为数不多的连接资源。接收排队的 3 个是无可更改的，但是发送排队却是可更改的，默认的值是 10 个，通过 MSG 指令可以更改为 40 个。修改这个值可以很好地改善控制器的

对外通信能力（如果有需求的话）。注意非连接限量这个值并不是从 10～40 之间的任意值，而是设置为 10 或 40，两种选择必居其一。

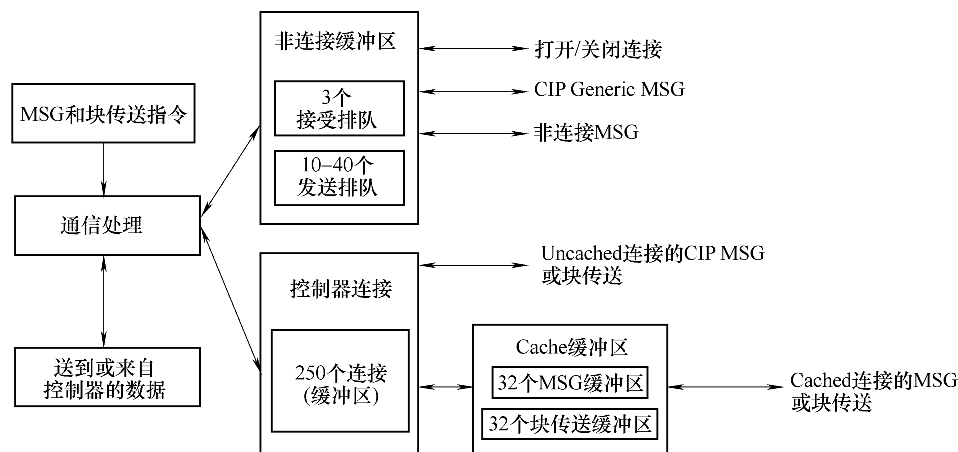


图 13-27　Logix 控制器通信资源示意图

在一个升级改造项目中，为了兼容传统产品，一个 Logix 控制器需要用 MSG 指令给 PLC5 发送大量的信息，经过一段时间的测试运行，信息数据传送经常失败，但凡发送失败的 MSG 指令提供的错误信息如图 13-28 所示，查阅在线帮助文件，扩展错误代码 16#0002_0301 的提示是：No buffer memory，这是告知非连接缓冲区空间不够使用，默认值 10 显然不能满足需要大量数据传输的需求，必须通过 MSG 指令的执行发送改变缓冲区大小的执行动作。



图 13-28　发送失败的 MSG 指令提供的错误信息

这里要留意，Logix 控制器与 PLC5 处理器之间的数据块传送不同于 Logix 控制器之间的数据块传送，Logix 控制器之间的数据传送操作是缓存连接（Cache Connections），受限于 32 个；Logix 控制器与 PLC5 之间发送数据块却是非连接缓冲，受限于 10 个或 40 个。我们这里要做的是如何从默认的 10 个增加到 40 个。

下面是专为修改非连接缓冲区大小而编制的一段梯级逻辑，很有实用价值。

首先，在控制器区域数据库中为 MSG 指令的执行创建 4 个短整字数组标签，如图 13-29 所示，这是 MSG 指令的服务性操作，访问控制器通信缓冲区需要用到的数据标签，里面将存放发送的设置信息或操作后读回的结果信息。

数据标签 Array_UCB_Set_Source 中设定更改非连接缓冲区发送连接限量大小的执行动作的数据；数据标签 Array_UCB_Get_Source 中设定获取非连接缓冲区发送连接限量大小的执行动作的数据，在数据表中分别应该设置的数据值如图 13-30 所示。这个访问对象所需要的执行代码须由专用的资料提供，不是任意设想的，只有指定的服务性代码数据才能让被操作对象识别并接受。

⊞-Array_UCB_Set_Source			SINT[8]
⊞-Array_UCB_Set_Destination			SINT[6]
⊞-Array_UCB_Get_Source			SINT[4]
⊞-Array_UCB_Get_Destination			SINT[10]

图 13-29　创建 4 个短整字数组标签

⊞-Array_UCB_Set_Source[0]	1
⊞-Array_UCB_Set_Source[1]	0
⊞-Array_UCB_Set_Source[2]	17
⊞-Array_UCB_Set_Source[3]	0
⊞-Array_UCB_Set_Source[4]	10
⊞-Array_UCB_Set_Source[5]	0
⊞-Array_UCB_Set_Source[6]	0
⊞-Array_UCB_Set_Source[7]	0

⊞-Array_UCB_Get_Source[0]	1
⊞-Array_UCB_Get_Source[1]	0
⊞-Array_UCB_Get_Source[2]	17
⊞-Array_UCB_Get_Source[3]	0

图 13-30　在两个标签中设定要发送的数据

创建短整字标签 UCB_Set_Value 和 UCB_Get_Value，将 Array_UCB_Set_Source 数组的第 4 个元素别名给 UCB_Set_Value，此为设置的非连接缓冲区发送连接限量的数值；将 Array_UCB_Get_Destination 数组的第 6 个元素别名给 UCB_Get_Value，此为读取的非连接缓冲区发送连接限量的数值，如图 13-31 所示，这两个量将用于比较判定是否相同，从而了解是否设置成功。

⊞-UCB_Set_Value	Array_UCB_Set_Source[4](C)

⊞-UCB_Get_Value	Array_UCB_Get_Destination[6](C)

图 13-31　设置和读取发送连接限量的两个标签

考虑到数组标签 Array_UCB_Set_Source 和 Array_UCB_Get_Source 初始数据设置的可靠性，如前面章节所介绍的，不妨执行 MOV 指令传送要设置的立即数来实现设定，编写的梯级逻辑如图 13-32 所示，将图 13-30 数据表中要设置的数据编程设定。

可以利用程序初次扫描来预设非连接缓冲区发送连接限量的大小，也可以由操作员在操作界面修改，如图 13-33 所示的梯级逻辑是为人机界面 PB 操作而编制的。当 PB 按下选择非连接缓冲区发送连接限量大小的按钮时，计时器开始计时，且计数器计数，有关操作延时并设置固定值我们曾在第 5 章讨论过，这里可以看到只有两个数据的设置，10 或 40，由计数器的奇偶数决定。如果将这个设定值反馈到人机界面，操作员可通过屏幕当即得知操作的结果。

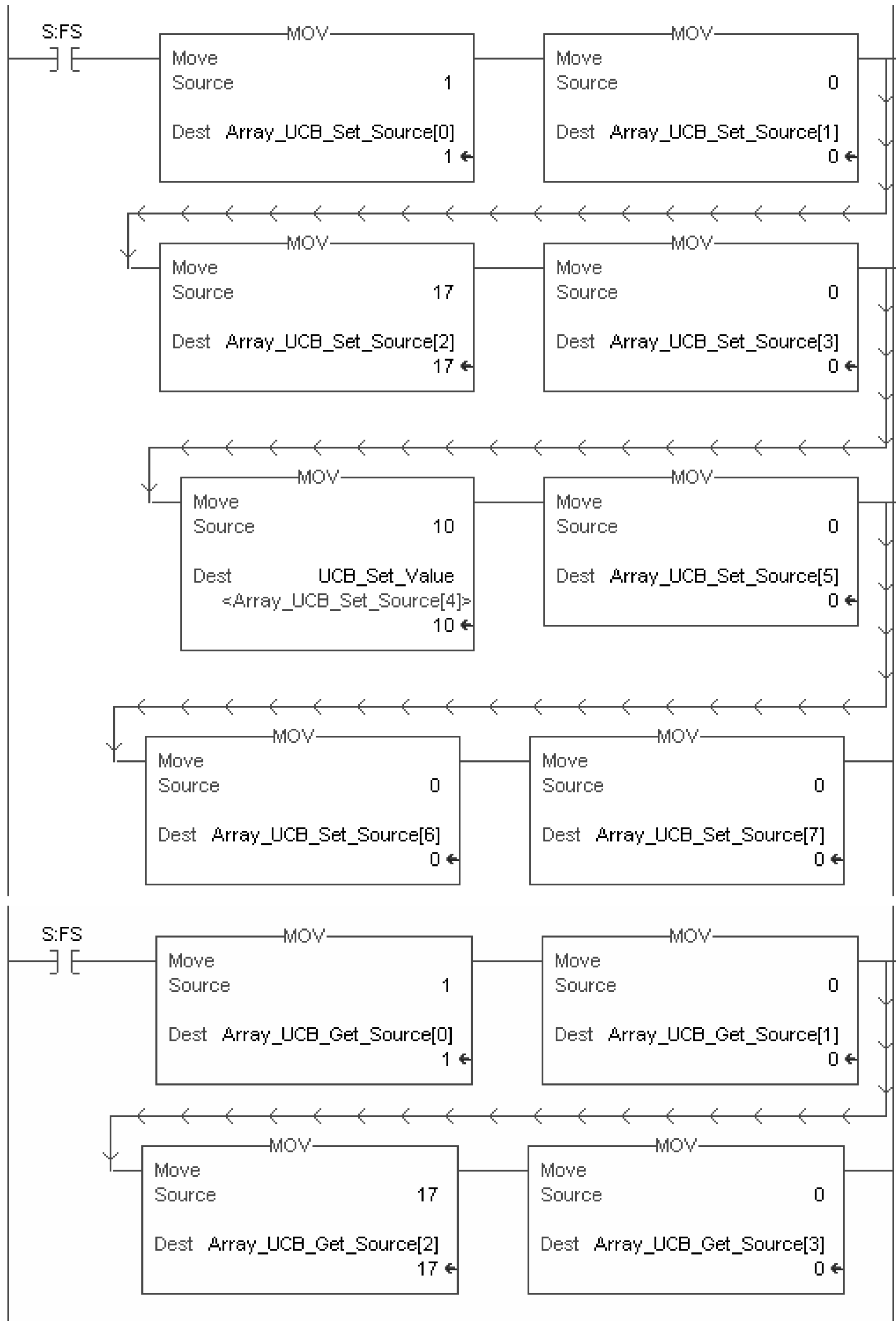


图 13-32　发送设定值的梯级逻辑

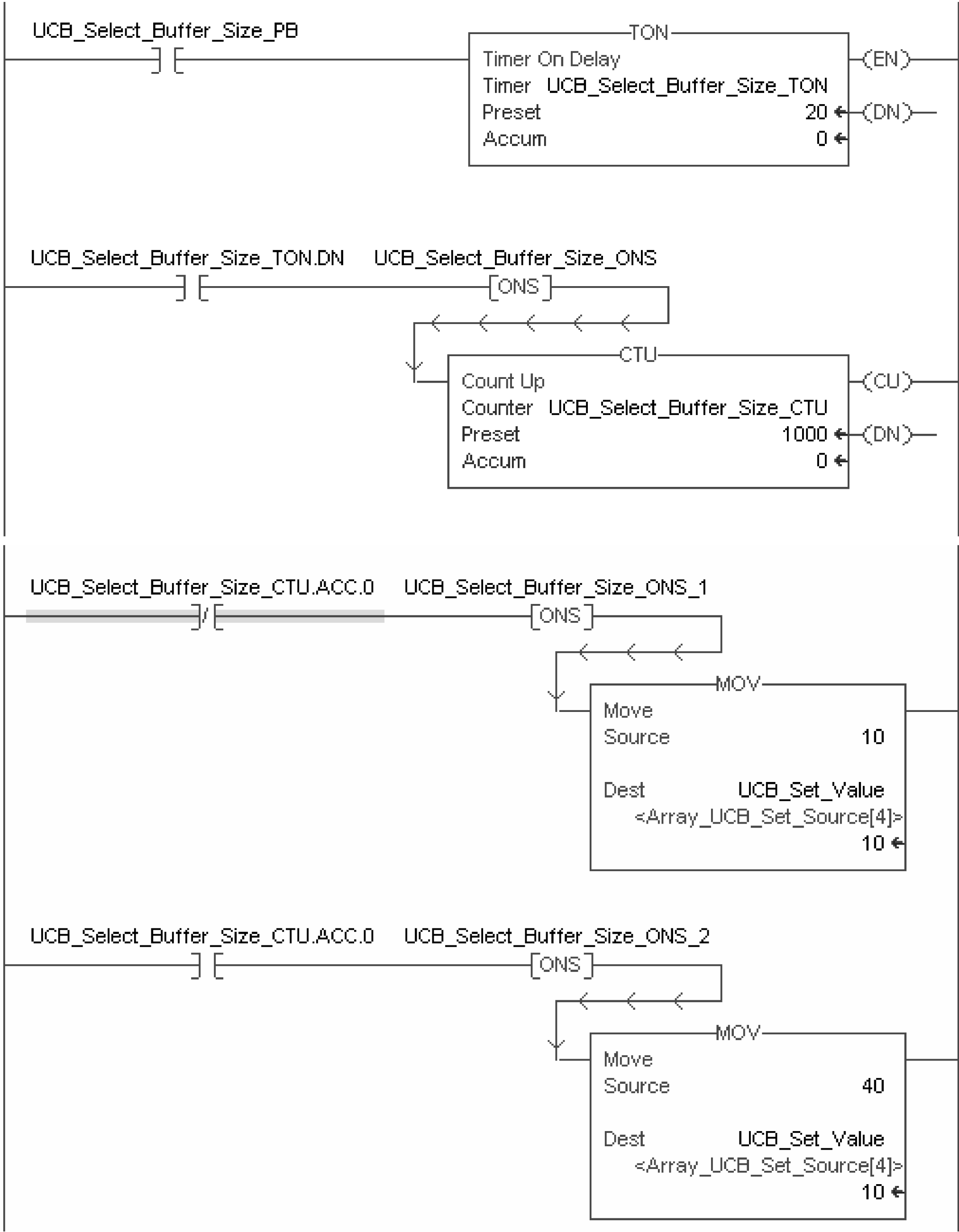


图 13-33　为人机界面 PB 操作而编制设定发送连接限量的梯级逻辑

梯级逻辑的编写如图 13-34 所示。当在人机界面 PB 上确认选定设置的数据是 10 或 40 之后，操作员按下设置的操作按钮，操作按钮 UCB_Set_PB 作为 MSG 的梯级条件，令设置非连接缓冲区发送连接限量值的 MSG 指令执行，MSG 指令执行后，完成预定的设置工作，完成位 UCB_Set. DN 置位，作为计时器 UCB_Get_TON 的梯级条件，在延时 0. 2s 之后，计时器完成位 UCB_Get_TON. DN 激发第二条 MSG 指令，即读取非连接缓冲区发送连接限量的值。

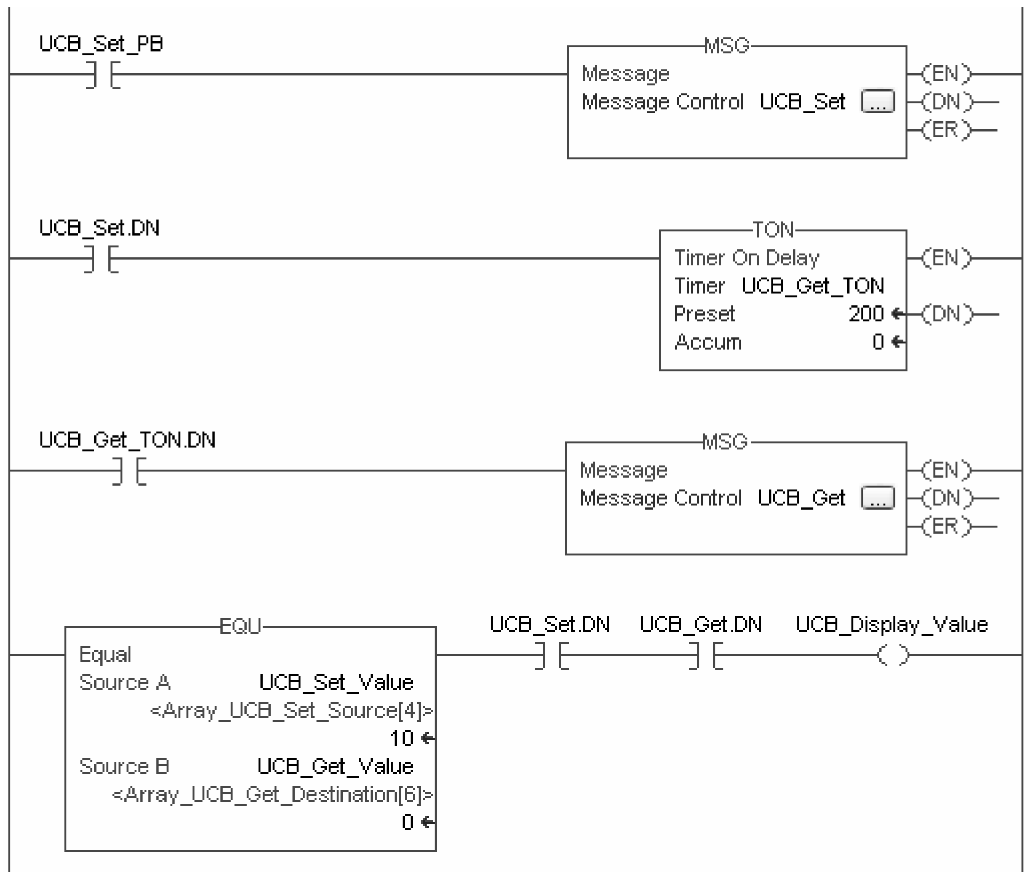


图 13-34　编写的梯级逻辑

执行设置非连接缓冲区发送连接限量值的 MSG 指令 UCB_Set，组态界面如图 13-35 所示，这是一个服务性代码的操作，请留意设定的服务类型代码等信息，这是相关资料提供的操作代码，为此才能被识别和接收。这条 MSG 指令的执行将送出 Array_UCB_Set_Source，这个之前设置好的数组数据将送出新组态的非连接缓冲区发送连接限量的数值，然后通过数组标签 Array_UCB_Set_Destination 返回设置信息。

UCB_Set 的 MSG 通信组态页面设置如图 13-36 所示，这是通过背板访问控制器自己的一条路径。

设置非连接缓冲区发送连接限量值的 MSG 指令 UCB_Set 完成之后，延时 0.2s，执行 MSG 指令 UCB_Get，其组态界面如图 13-37 所示，这是一个服务性代码的操作，请留意设定的服务类型代码等信息，这是相关资料提供的操作代码，为此才能被识别和接受。这条 MSG 指令的执行将送出 Array_UCB_Get_Source 的操作，并通过数组标签 Array_UCB_Get_Destination 读回新组态的非连接缓冲区发送连接限量的数值。

UCB_Get 的 MSG 通信组态页面设置如图 13-38 所示，这是通过背板访问控制器自己的一条路径。

最后一个梯级的执行，将非连接缓冲区发送连接限量值的设定值 UCB_SetUCB_Set_

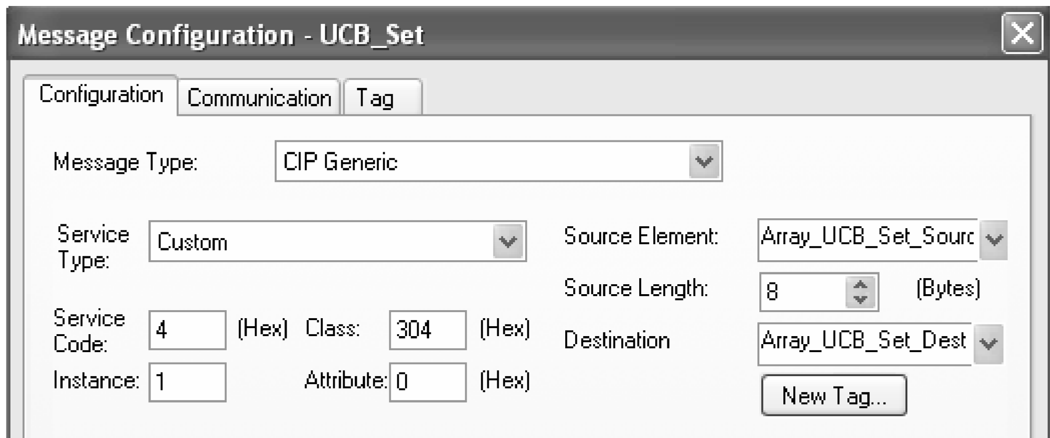


图 13-35　设置发送连接限量值的组态

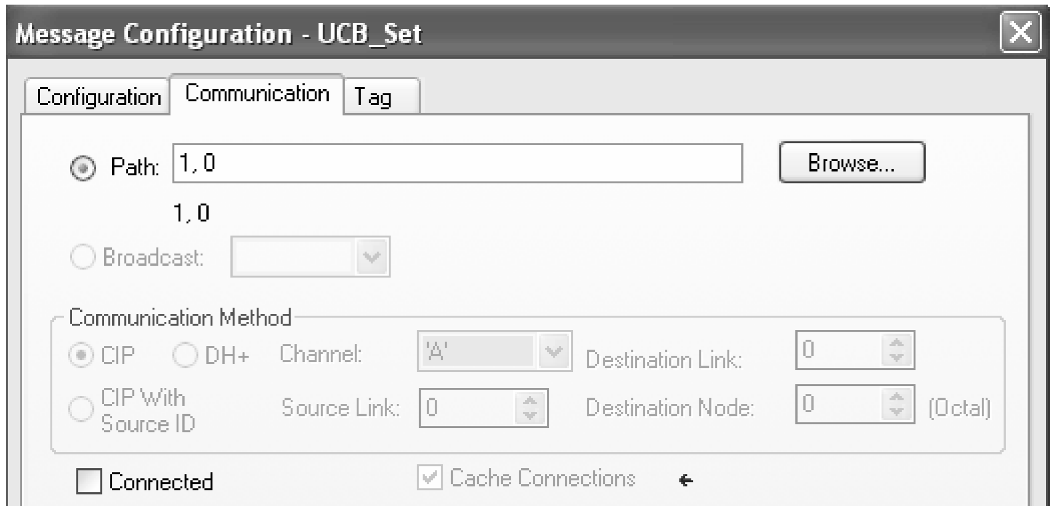


图 13-36　设置发送连接限量值的通信组态

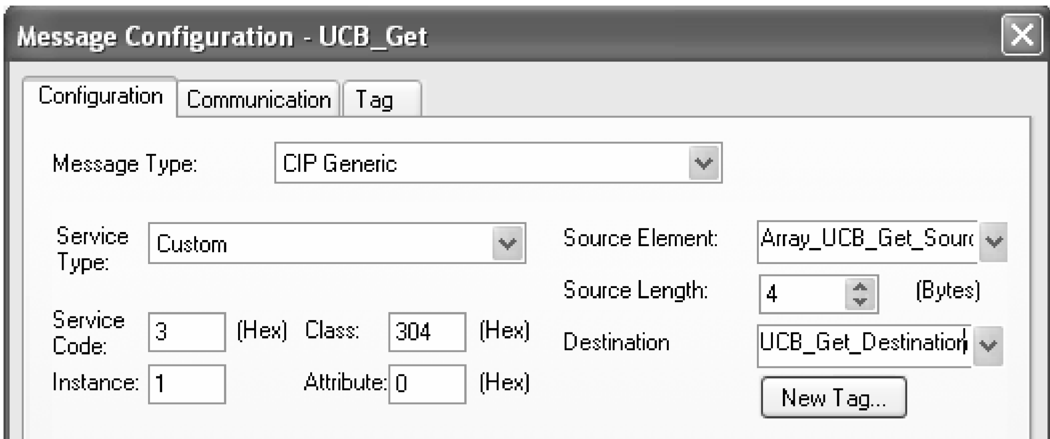


图 13-37　读取发送连接限量值的组态

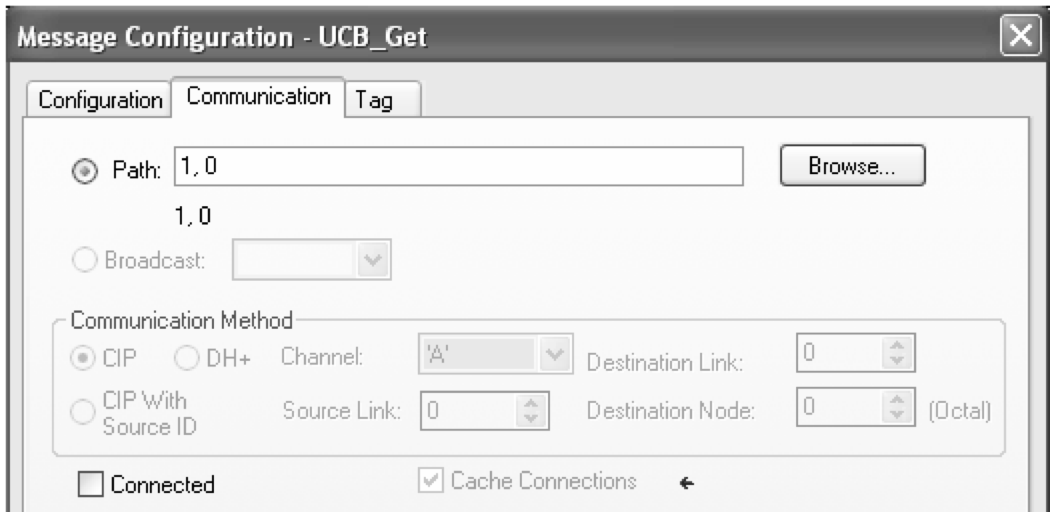


图 13-38 读取发送连接限量值的通信组态

Value 和非连接缓冲区发送连接限量值的返回值 UCB_Get_Value 进行比较，确认它们相等，且两条 MSG 指令都已完成，核实正确后给出信息显示，标签 UCB_Display_Value 被置位，提示设置成功完成。

完整的梯级逻辑如图 13-39 所示，运行测试结果正常，将非连接缓冲区发送排队连接的 10 个限量成功地改为 40 个限量。

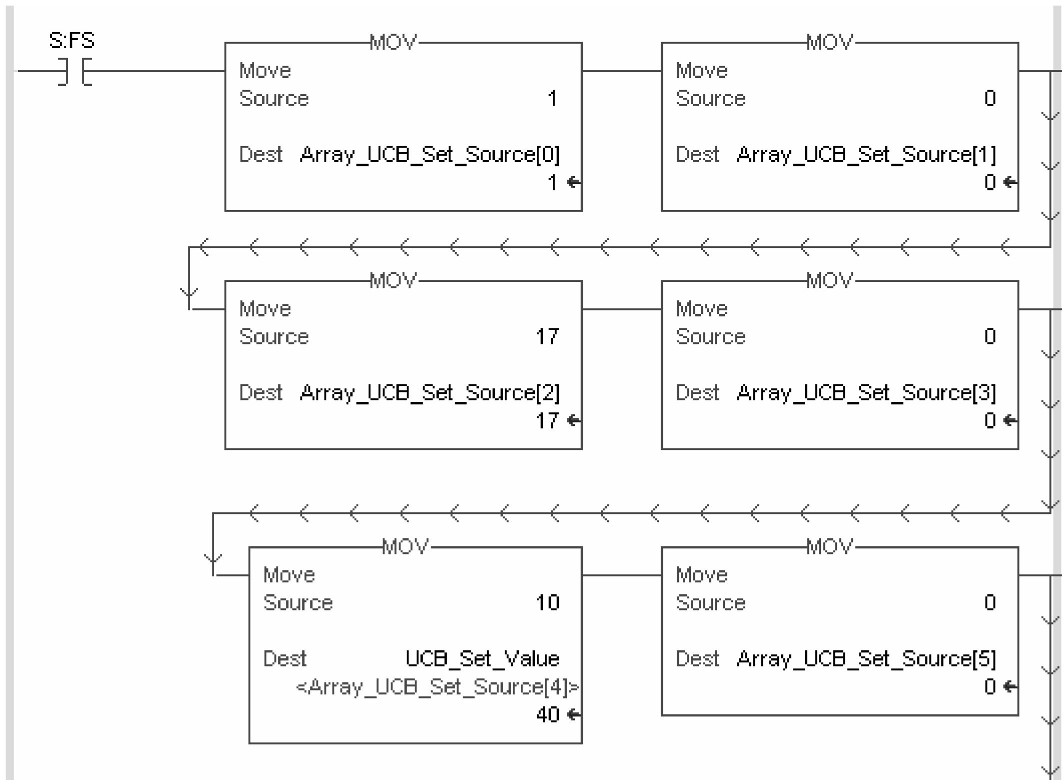


图 13-39 发送连接限量设定完整的梯级逻辑

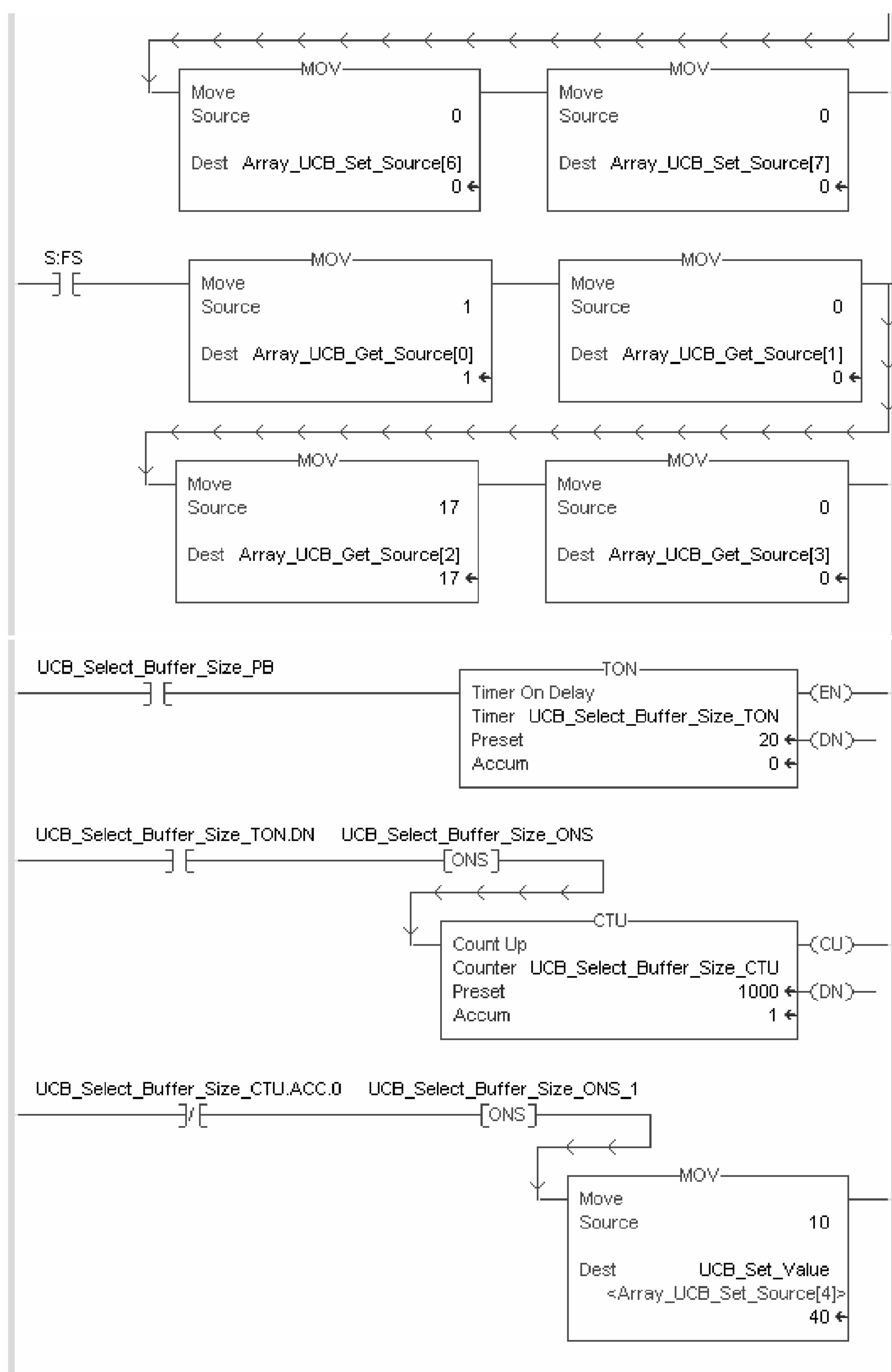


图 13-39　发送连接限量设定完整的梯级逻辑（续）

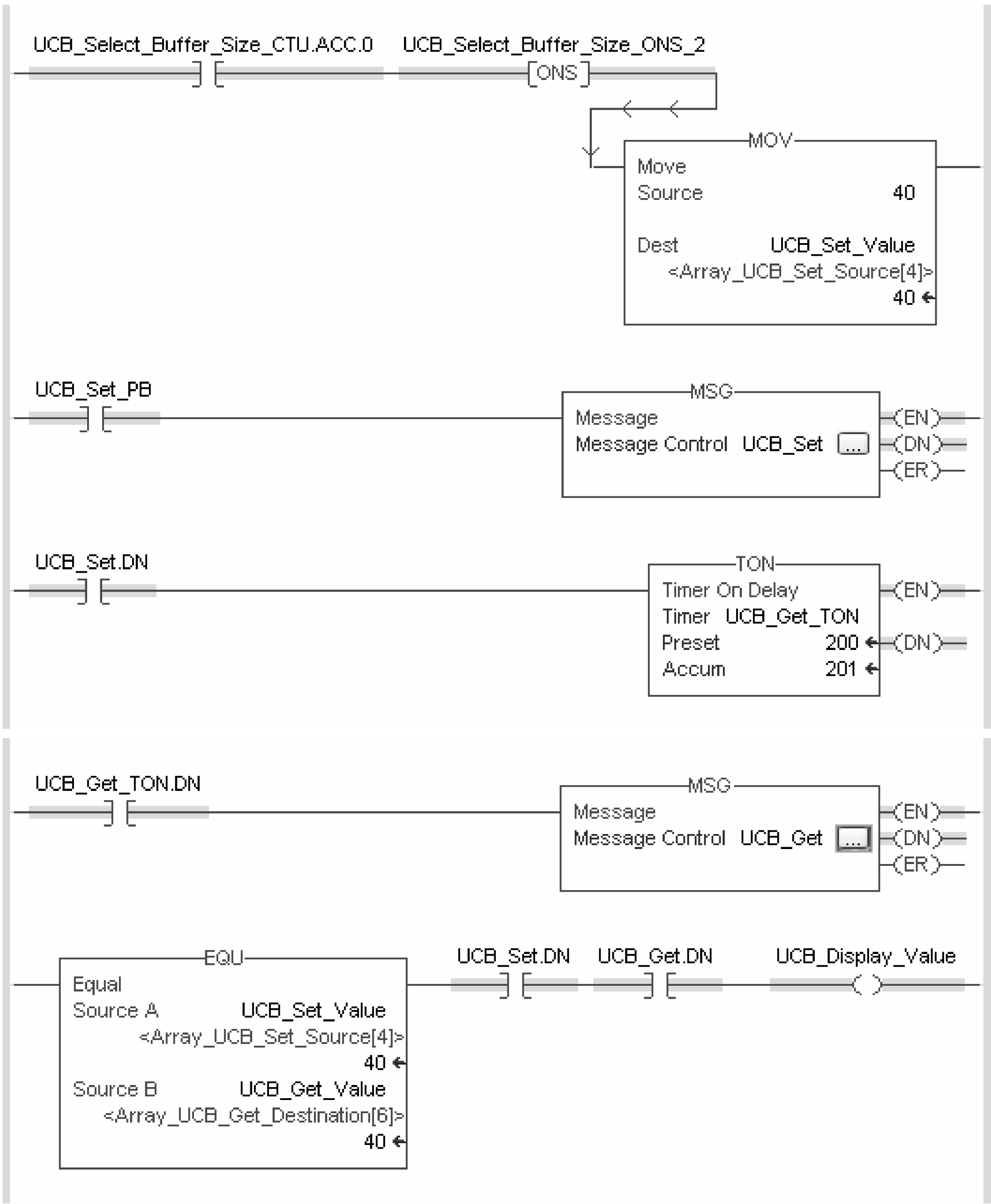


图 13-39　发送连接限量设定完整的梯级逻辑（续）

第14章

系统设置和系统状态指令 SSV/GSV 的运用

对于功能模块化结构的 Logix 系列的控制器所构成的控制系统，每个模块或设备都是智能的，都具有通信能力，能够交换信息数据。控制器可以设置或监视控制系统中所有的硬件设备和控制器自身的工作状态。由于每个控制系统结构都不尽相同，控制器并不设立固定的监视信息，而是由用户自己决定要设置或监视哪些信息，编写程序去设置或获取系统的信息，并利用这些信息为控制系统服务。

设置系统信息和获取系统信息的工作要使用 SSV 指令和 GSV 指令来完成，SSV 指令（Set System Value），设置系统值；GSV 指令（Get System Value），获取系统值，SSV 指令和 GSV 指令都是对系统进行访问的。

SSV 指令和 GSV 指令要访问的对象如下：

- AXIS：伺服控制的大量信息；
- CONTROLLER：控制器用于通信管理的 CPU 百分比值的信息；
- CONTROLLERDEVICE：控制器模块产品及状态信息；
- CST：协调系统时间的状态及当前值的信息；
- DF1：串口 DF1 通信协议的组态和状态信息；
- FAULTLOG：控制器主要故障和次要故障信息；
- MESSAGE：MSG 指令执行状态信息；
- MODULE：模块的状态信息；
- MOTIONGROUP：伺服模块运控组名称的信息；
- PROGRAM：程序执行状态信息；
- ROUTINE：例程执行状态信息；
- SERIALPORT：串口通信组态信息；
- TASK：任务执行状态和组态信息；
- WALLCLOCKTIME：控制器系统日期时间信息。

以上列举的访问对象有些是只读信息，仅可使用 GSV 指令，有些是可读可写信息，GSV 指令和 SSV 指令均可使用，例如控制器系统的日期和时间就是可读可写信息。下面我们将以此为例，编写有关的梯级逻辑，初步了解 SSV 指令和 GSV 指令的运用。

被访问的对象具有不同的意义和内容，因而有不同的数据结构，在 SSV/GSV 的指令参数中，必须使用符合对象结构的数据标签，所以大多情况下，用户一定要自己创建相应的结构数据和建立对应的数据标签，并对结构数据的元素加以说明，一目了然信息的含义。如果对要访问的对象数据结构不是十分清楚，要在在线帮助文件中寻找详细信息。

14.1 访问系统日期时间

下面我们以设定和读取控制器系统日期时间为例，体会访问系统信息的过程。于在线帮助文件中，我们可以找到如图 14-1 所示的详细说明，所列举的有对象内容、数据类型、指令类别和有效数据的说明。

Attribute	Data Type	Instruction	Description	
LocalDateTime	DINT[7]	GSV SSV	The controller local date and time in human readable format, as follows.	
			Value	**Meaning**
			DINT[0]	Year
			DINT[1]	Month (1...12)
			DINT[2]	Day (1...31)
			DINT[3]	Hour (0...59)
			DINT[4]	Minute (0...59)
			DINT[5]	Seconds (0...59)
			DINT[6]	Microseconds (0...999,999)

图 14-1　控制器日期时间的详细说明

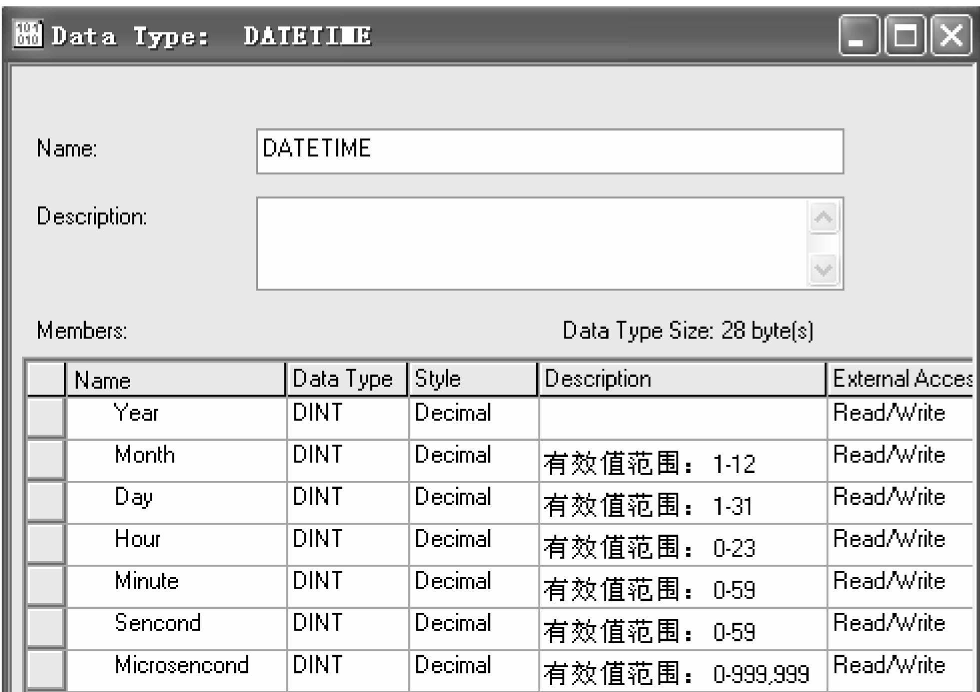
被访问的对象日期时间是一个 7 个双整字的结构数据或数组，对应访问对象结构，创建一个在用户自定义日期时间的结构数据，如图 14-2 所示。子元素的命名都有明确的含义，注意年、月、日参数的数据有效范围，这是不能为 0 的数据，如果不进行任何设置，意味着设置为 0，也就是不在有效数据范围之内。

子元素的命名必须是 ASCII 码所能表达的字母，因为系统执行调用识别所需；说明却可以适用任何文字，譬如说中文，因为这是文档的文字，系统执行时不会访问。

在程序数据库中，创建日期时间的数据标签，标签 DateTime_Get 用来获取日期时间数据；标签 DateTime_Set 用来设置日期时间数据。两个标签选择的数据类型是刚刚创建的用户自定义数据结构 DATEYTIME，如图 14-3 所示。可见，展现标签的子元素正是系统日期时间的数据结构。

编写的梯级逻辑如图 14-4 所示。SSV 的指令只在设置的时候执行，所以需要梯级条件来决定，GSV 指令则不断地执行并获得当前值，如果不需要跟随系统日期时间，可加上梯级条件予以限制，以减少程序运行时间。

在 DateTime_Set 结构数据标签中设置日期和时间的数值，一定要按照有效值的范围设置，如果设置的数据在允许范围之外，SSV 指令仍然执行，但无效数据送到控制器时会被拒绝，且没有任何错误的提示，我们能看到的仅仅是日期时间没有被设置成功。例如年份、月份和日的设置值是不能为零的，如果像平时我们测试数据块传送那样，只设最前面一个，其余未被设定的值是 0，这恰恰是有效值范围之外的。


Data Type:　DATETIME

Name: DATETIME

Description:

Members:　　Data Type Size: 28 byte(s)

Name	Data Type	Style	Description	External Acces
Year	DINT	Decimal		Read/Write
Month	DINT	Decimal	有效值范围：1-12	Read/Write
Day	DINT	Decimal	有效值范围：1-31	Read/Write
Hour	DINT	Decimal	有效值范围：0-23	Read/Write
Minute	DINT	Decimal	有效值范围：0-59	Read/Write
Sencond	DINT	Decimal	有效值范围：0-59	Read/Write
Microsencond	DINT	Decimal	有效值范围：0-999,999	Read/Write

图 14-2　用户自定义日期时间的结构数据

⊟-DateTime_Get		DATATIME	
⊞-DateTime_Get.Year		DINT	Decimal
⊞-DateTime_Get.Month		DINT	Decimal
⊞-DateTime_Get.Day		DINT	Decimal
⊞-DateTime_Get.Hour		DINT	Decimal
⊞-DateTime_Get.Minute		DINT	Decimal
⊞-DateTime_Get.Sencond		DINT	Decimal
⊞-DateTime_Get.Microsencond		DINT	Decimal
⊟-DateTime_Set		DATATIME	
⊞-DateTime_Set.Year		DINT	Decimal
⊞-DateTime_Set.Month		DINT	Decimal
⊞-DateTime_Set.Day		DINT	Decimal
⊞-DateTime_Set.Hour		DINT	Decimal
⊞-DateTime_Set.Minute		DINT	Decimal
⊞-DateTime_Set.Sencond		DINT	Decimal
⊞-DateTime_Set.Microsencond		DINT	Decimal

图 14-3　用户自定义日期时间数据结构标签

对控制器日期时间访问的 SSV 指令和 GSV 指令执行之后，数据表的监视数据显示如图 14-5 所示。

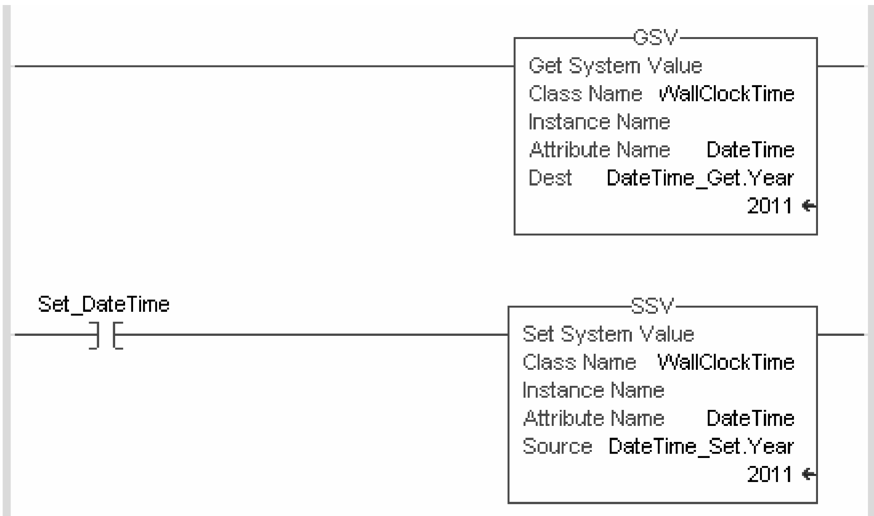


图 14-4　设定和读取日期时间的梯级逻辑

⊟ DateTime_Get	{...}	
⊞ DateTime_Get.Year	2011	Decimal
⊞ DateTime_Get.Month	5	Decimal
⊞ DateTime_Get.Day	6	Decimal
⊞ DateTime_Get.Hour	17	Decimal
⊞ DateTime_Get.Minute	3	Decimal
⊞ DateTime_Get.Sencond	1	Decimal
⊞ DateTime_Get.Microsencond	958766	Decimal
⊟ DateTime_Set	{...}	
⊞ DateTime_Set.Year	2011	Decimal
⊞ DateTime_Set.Month	5	Decimal
⊞ DateTime_Set.Day	6	Decimal
⊞ DateTime_Set.Hour	17	Decimal
⊞ DateTime_Set.Minute	2	Decimal
⊞ DateTime_Set.Sencond	0	Decimal
⊞ DateTime_Set.Microsencond	0	Decimal

图 14-5　日期时间数据表的监视数据显示

14.2　访问控制器的任务和程序

GSV/SSV 指令还可以访问控制器任务执行的状态，例如，当控制器设定多个任务时，任务调用有可能出现交迭的情况，这将使得执行代码不能完成，埋下隐患，所以需要对任务交迭状态监控。控制器任务监视页面有一个任务交迭计数器，但是我们不可能时时在线查看，编写梯级逻辑获取当前任务的交迭计数值便可以对当前任务的执行情况及时了解。

于在线帮助文件中，找到如图 14-6 所示的详细说明，交迭计数对应的是一个双整字，创建双整字标签 Task_OverlapCount，交迭计数可以被 SSV 指令和 GSV 指令访问，是可设置可读取的访问对象。

Attribute	Data Type	Instruction within Standard Task	Instruction within Safety Task
OverlapCount	DINT	GSV SSV	GSV SSV

图 14-6　交迭计数的详细说明

编写的梯级逻辑如图 14-7 所示。这里编写的两个梯级仅仅是针对交迭计数，GSV 指令读取的计数值如果大于 0，是应该做相应的处理的，处理完毕的交迭计数需要清除，SSV 指令则完成了交迭计数的清除工作。

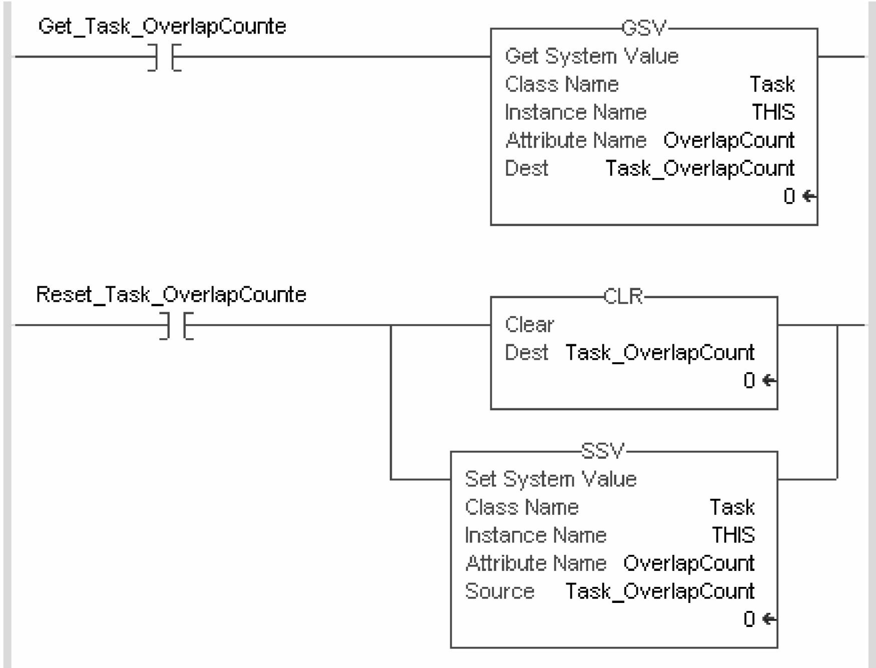


图 14-7　访问任务交迭计数的梯级逻辑

梯级条件 Get_Task_OverCounte 给定时，使能 GSV 指令，访问控制器中的任务。THIS 被称为关键字，是系统直接访问标签，无须创建，这里 THIS 指的是执行 GSV/SSV 指令梯级所在的当前任务。GSV 指令的执行获取了监视信息交迭计数的当前值，这个值随着任务出现交迭状态而累加，系统记录的交迭状态计数一旦被设置，是不会随着警告的状态已经消失而消失的，必须执行 SSV 指令去清除。此处的 SSV 指令不是用来设定值而是用来清除交迭计数的值，即将 0 这个特殊的值送到任务迭加计数单元，达到清零的目的。

也可以访问控制器中的程序，例如获取某个程序的最后扫描时间和最大扫描时间。在控

制器程序监视页面，这个监视信息也是可以读到的。如果需要提供程序数据，就需要运用 GSV 指令来访问了。

于在线帮助文件中，找到如图 14-8 所示的详细说明。最后扫描时间和最大扫描时间对应的都是一个双整字，创建双整字标签 Get_LastScanTime 和 Get_MaxScanTime，扫描时间监视参数可以被 SSV 指令和 GSV 指令访问。

Attribute	Data Type	Instruction within Standard Task	Instruction within Safety Task
LastScanTime	DINT	GSV SSV	None
MaxScanTime	DINT	GSV SSV	None

图 14-8　程序最后扫描时间和最大扫描时间的详细说明

编写的梯级逻辑如图 14-9 所示。Logix 控制器允许将这样的输出指令串联在同一个梯级，常常可以看到将 GSV 指令或 SSV 指令串联在一个梯级的做法。这里 THIS 指的是执行 GSV 指令梯级所在的当前程序。

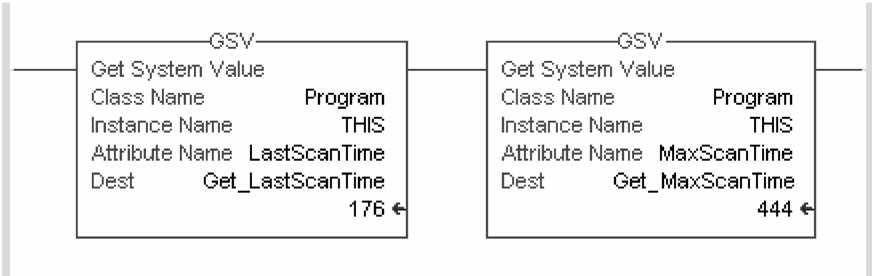


图 14-9　获取程序扫描时间的梯级逻辑

当使用 SSV 指令对程序进行最后扫描时间和最大扫描时间操作时，就意味着清除了。对于最大扫描时间的清除，相当于复位的操作，但是最后扫描时间立刻就被新的时间代替，SSV 指令的操作几乎没有意义。

14.3　访问控制系统中的模块或设备

执行 GSV 指令还可以获取模块的故障代码，这是一种非常典型的对模块的访问操作。请注意，不是所有的模块都能被控制器读取故障信息，只有模块与控制器之间原本就存在通信关系的智能模块才能交换模块的状态信息。

于在线帮助文件中，找到如图 14-10 所示的详细说明。模块故障代码是整型字，分别为两个被访问的模块创建整型字标签 DInput_DOutput_FaultCode 和 AInput_AOutput_FaultCode，模块故障代码是只读状态信息，只能被 GSV 指令访问，是仅可读取的访问对象。

Attribute	Data Type	Instruction	Description
FaultCode	INT	GSV	A number which identifies a module fault, if one occurs.

图 14-10　模块故障代码的详细说明

梯级逻辑的编写如图 14-11 所示。梯级条件成立时，从指定模块中获取故障代码，存放在创建的故障代码数据标签中。模块指定可以从浏览 I/O 组态中找到，也即跟控制器保持通信交换数据的模块。

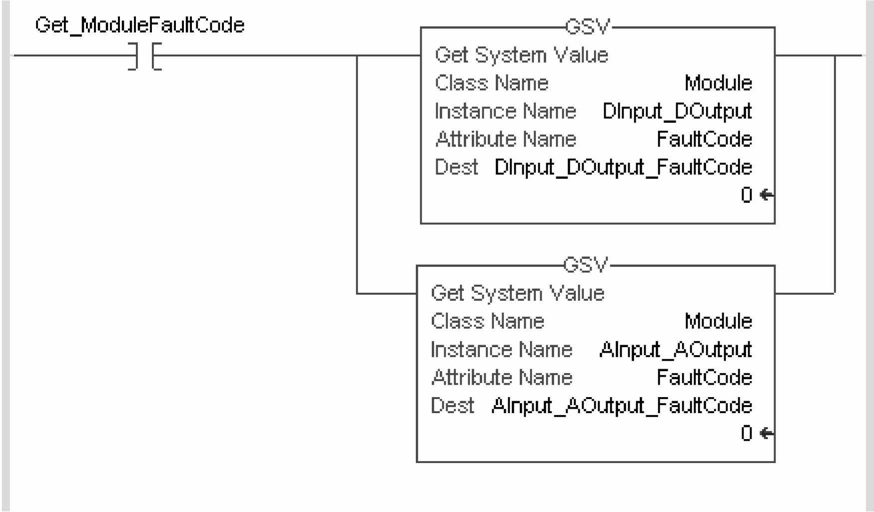


图 14-11　访问模块故障代码的梯级逻辑

像所有的计算机系统一样，控制器对外通信的资源无非是空间和时间，前面说到的 MSG 指令章节中，我们曾经讨论如何修改控制器通信缓冲区，这是对控制器通信资源空间的调整，下面我们再来看看对控制器通信资源时间的调整。

Over head Time Slice 是一个直接关系到控制器的对外通信能力的参数设置，这个设定值的真正含义是控制器逻辑 CPU 连续扫描执行代码和对外通信的比值关系，即对外通信占用逻辑 CPU 工作的百分比。早期版本的控制器对这个参数是十分敏感的，这个设置值偏小将影响到控制器的通信能力，偏大则影响执行代码的扫描速度，直到 16 版本之后，这个问题才得到较好的解决，增加了新的选项，用户可以选定当不使用 Overhead Time Slice 时，逻辑 CPU 扫描执行代码，将通信富余的时间用来扫描执行代码。

设定这个参数通常在编程软件的控制器属性中设置，如图 14-12 所示。

为控制器选定一个合适的 Overhead Time Slice 百分比值，是控制系统现场调试要做的工作之一，在测试阶段，不妨手动在外部逐步调整，以求得最佳设定值。

于在线帮助文件中，找到如图 14-13 所示的详细说明，Time Slice 是整型字，为被访问的对象创建整型字标签 OverHeadTimeSlice 和 OverHeadTimeSlice_New，这个参数可以被 SSV 指令和 GSV 指令访问。

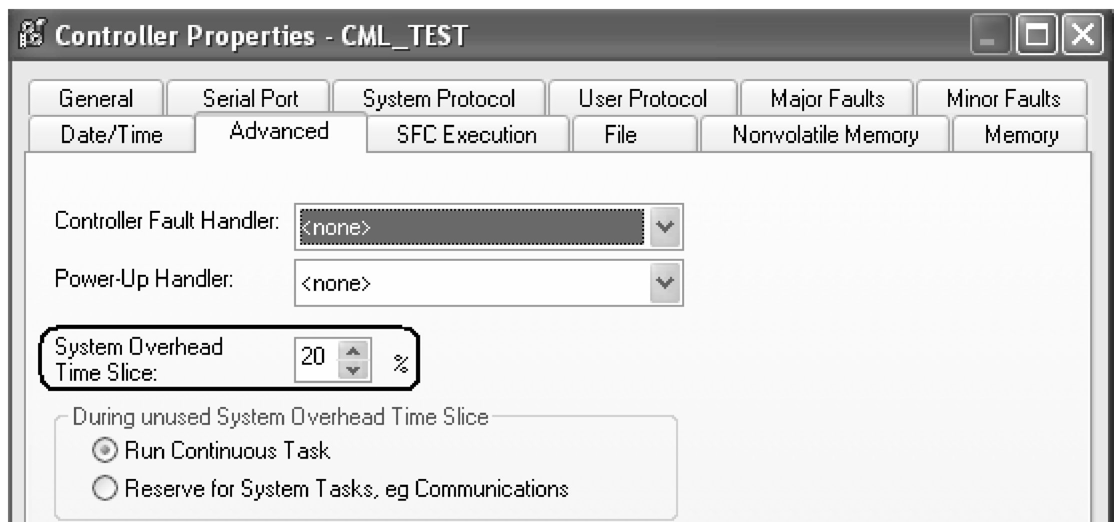


图 14-12　在编程软件的控制器属性中设置参数

Attribute	Data Type	Instruction	Description
TimeSlice	INT	GSV SSV	Percentage of available CPU that is assigned to communications. Valid values are 10-90. This value cannot be changed when the controller keyswitch is in the Run position.

图 14-13　通信时间片段的详细说明

如果要外部手动设置 Overhead Time Slice 参数，可以由操作人员通过模拟量旋钮设置这个系统的内部参数，控制器则以指令执行来实现，编写的梯级逻辑如图 14-14 所示。

运用事先编制好的 Add_On 模拟量定标指令（后续章节将详细讨论），将指定通道的模拟量 A/D 转换后的数据范围 0 ~ 31104 定标为设定所需的 0 ~ 10 的范围值，这个数值范围将对应要设置的 10% ~90%，一共 9 个设置范围。

执行 GSV 指令获取当前 Overhead Time Slice 值，这个当前值将作为确定设置怎样的比值的判断条件，当前值与它一致则不进行设置操作。

连续 9 个梯级都是相同的梯级逻辑关系，用以设定 9 个不同的 OverheadTimeSlice 值。首先通过 LIM 指令确定设置范围，然后对比当前 OverheadTimeSlice 是否不等于设置范围，同时满足这两个条件，则梯级条件成立，传送相应设置值到缓冲数据标签 Overhead Time Slice_New，最后由 SSV 指令送至控制器系统。如果当前 Overhead Time Slice 已经是欲设定的值，则无需操作 SSV 指令再次设置。

Overhead Time Slice 的设定值通常在 20% ~30%，在某些特定的情况（如冗余系统）下可能会高达 50%，像采用 90% 这样的极端情况是极为少见的。过高的设定值将使得梯级逻辑的扫描时间变得很慢。

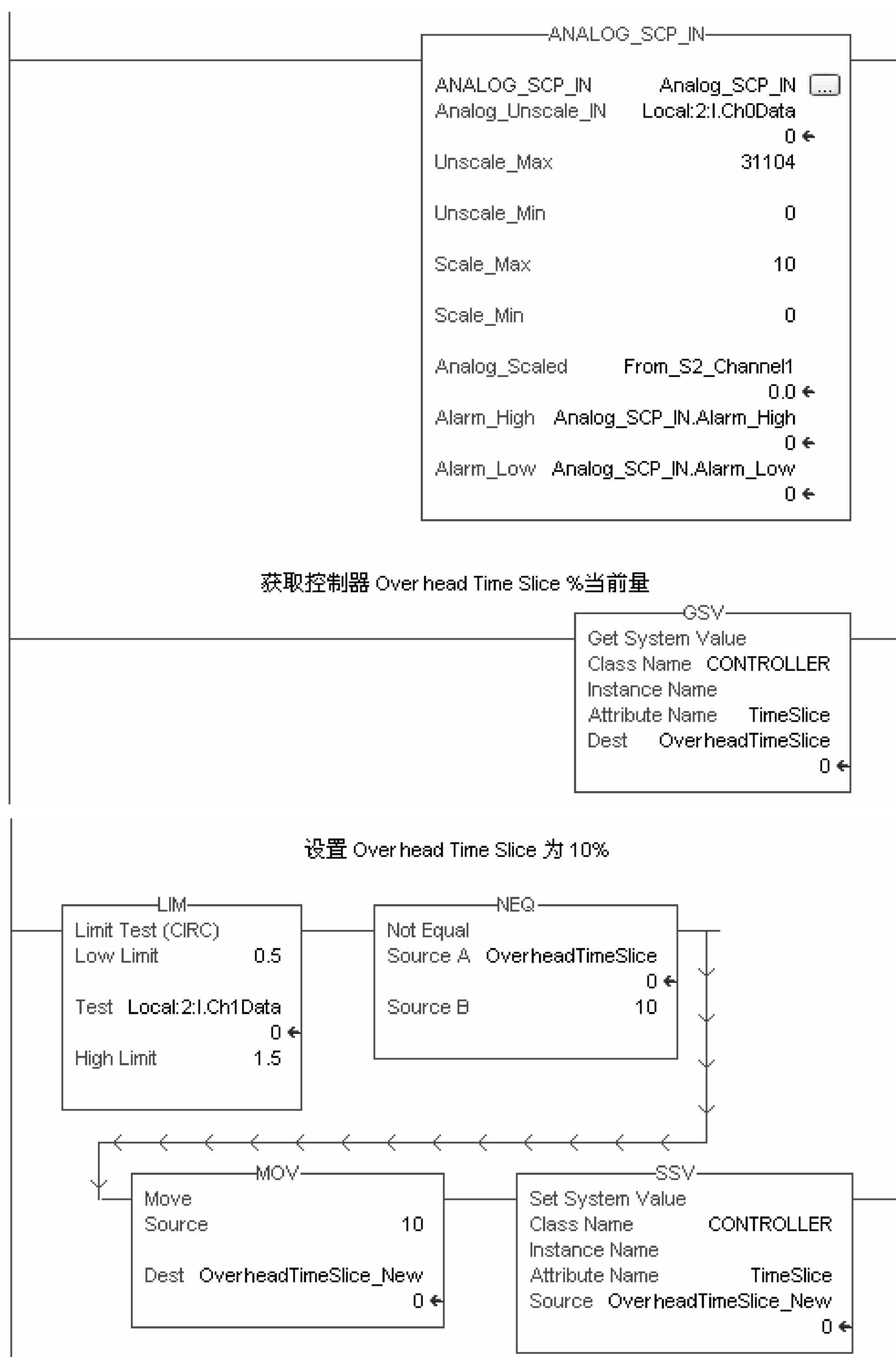


图 14-14　调整通信时间片段的梯级逻辑

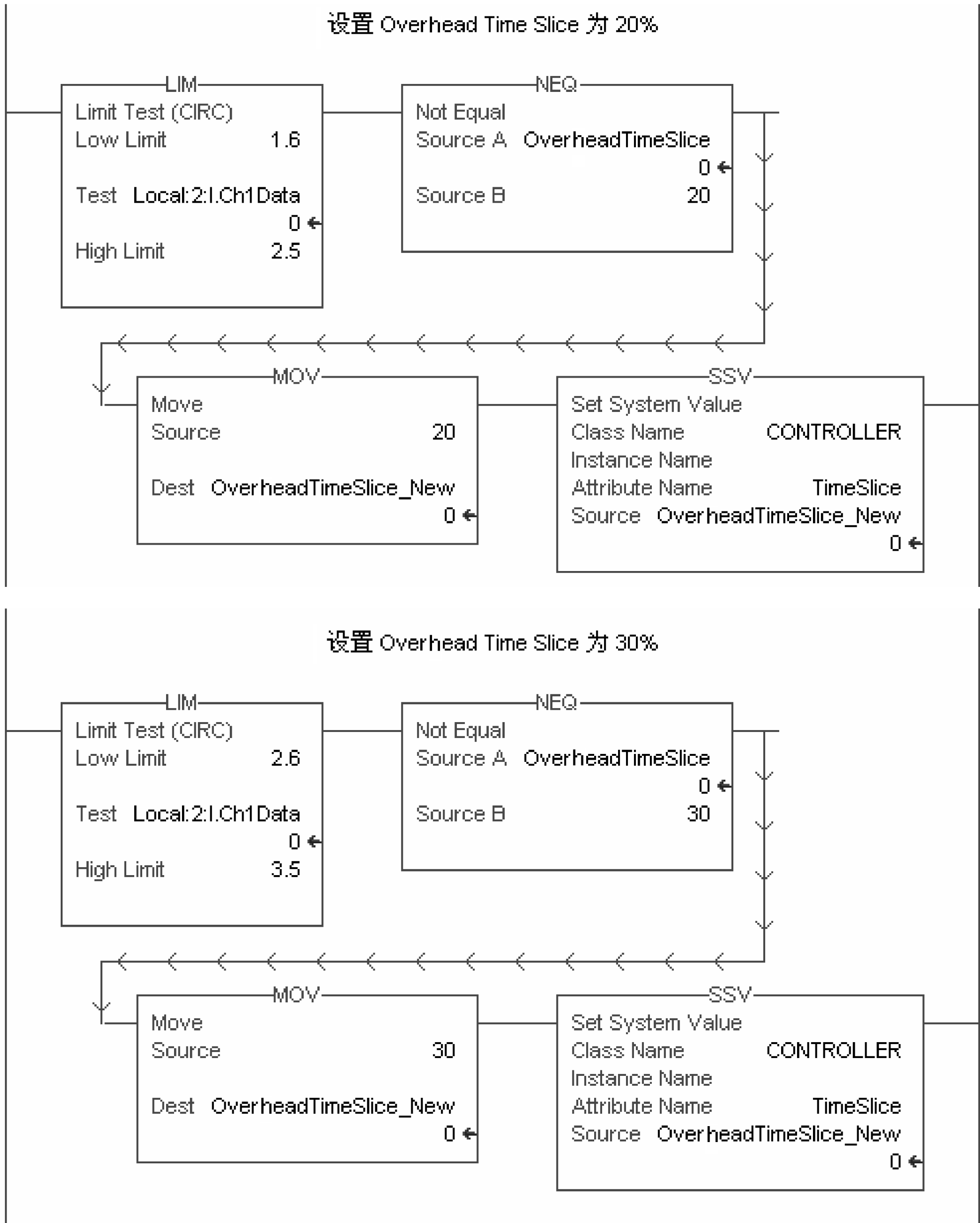


图 14-14　调整通信时间片段的梯级逻辑（续）

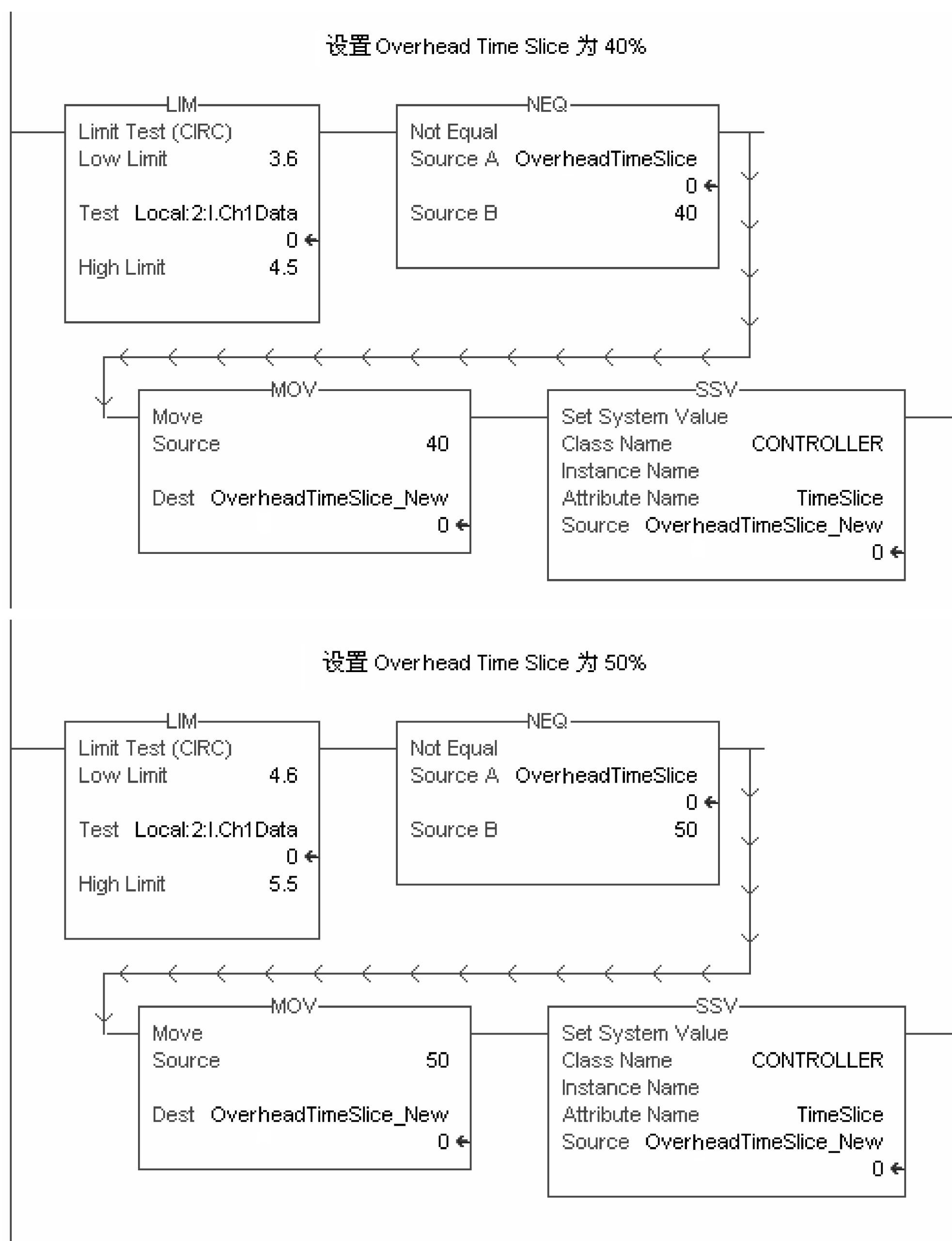


图 14-14　调整通信时间片段的梯级逻辑（续）

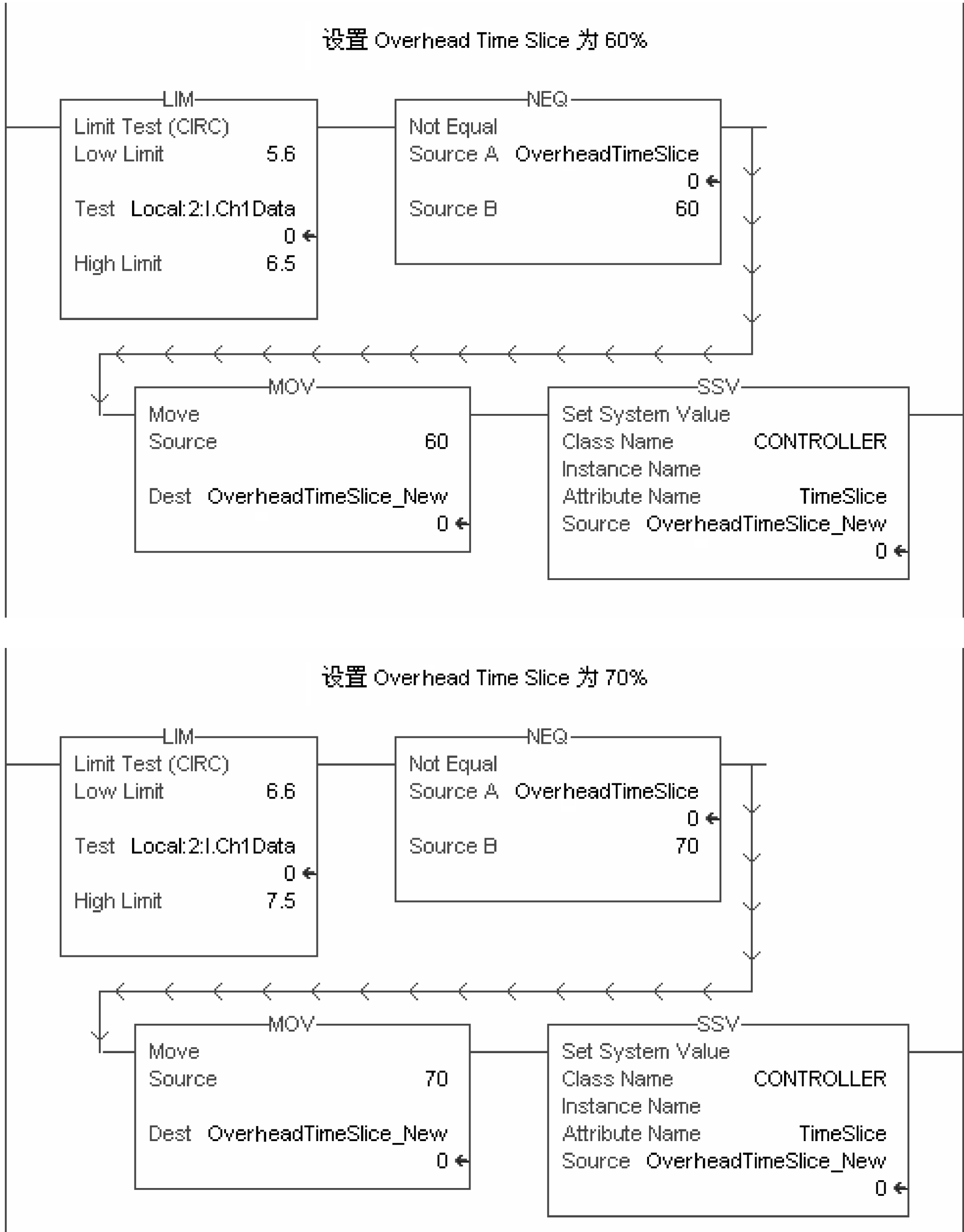


图 14-14　调整通信时间片段的梯级逻辑（续）

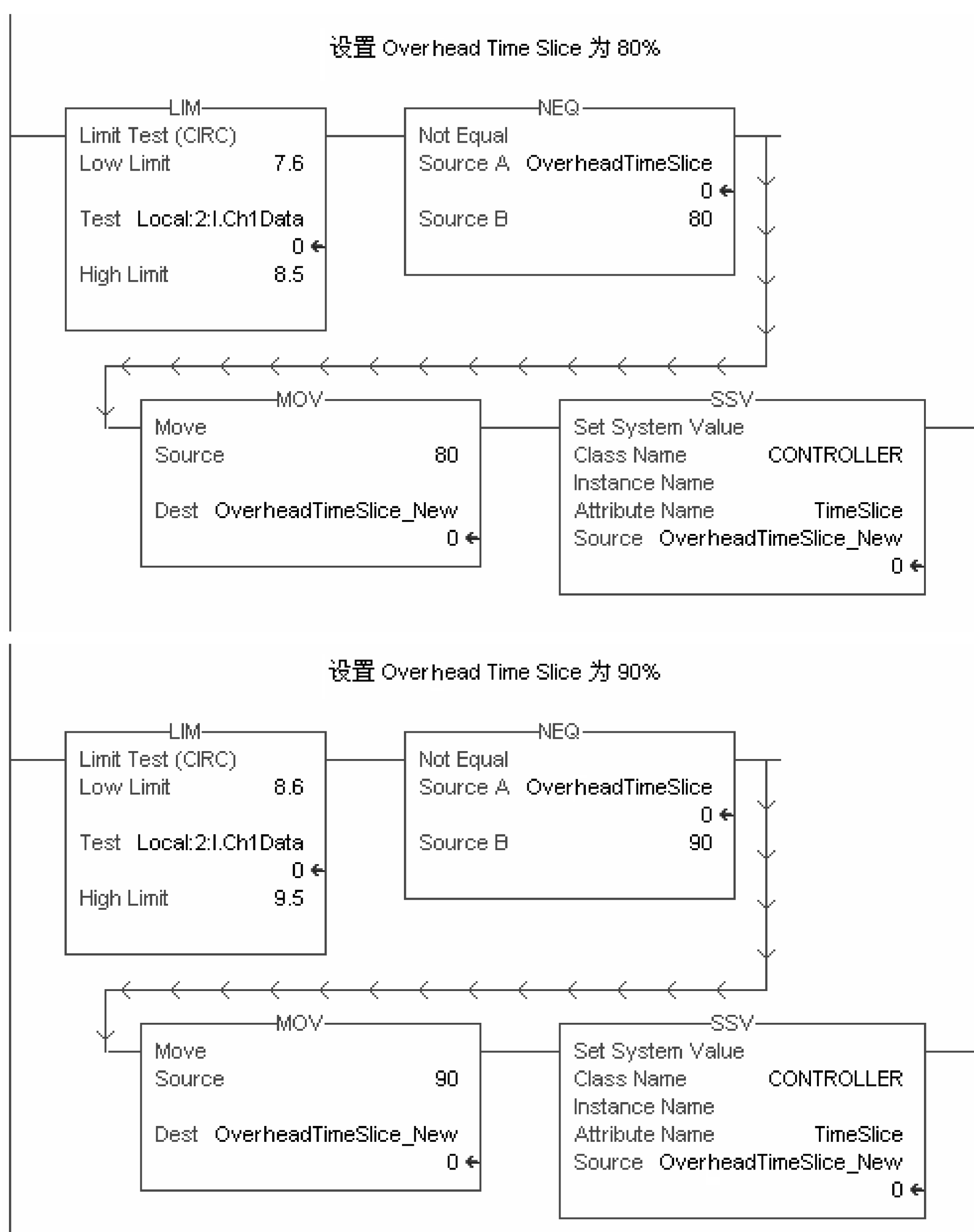


图 14-14　调整通信时间片段的梯级逻辑（续）

你可能注意到了，我不厌其烦地将帮助文件中访问对象的结构数据的详细说明列举出来，其目的就是提醒你要特别地关注面对的这个信息是什么内容，具有什么样的数据类型，有效的数据是什么。因为 SSV/GSV 指令执行访问失败后，基本是没有反馈信息提供给你的。

MSG 指令和 SSV/GSV 指令都可以对控制系统中的模块或设备操作，它们的操作有什么区别呢？

SSV/GSV 指令操作的是组态信息和状态信息，设置模块或设备中的组态信息，获取模

块或设备中的状态信息。这些信息在用 SSV/GSV 指令操作时，借助于系统预定义的结构数据来表达，具有固定的形式，同样的信息还可以出现在编程软件固有的组态或状态界面中。这其实是一套数据的两种不同的表达形式，SSV/GSV 指令的结构数据所表达的信息可以直接的被程序引用并为实施逻辑监视提供数据，也可以为人机界面的监视提供数据，且更为详细。

MSG 指令对模块或设备操作时，属于服务性操作，是具体的命令式的操作，发送的是命令动作，它所执行的正是组态或状态界面上我们用鼠标点击的那些按钮，那些按钮下隐藏的命令操作与 MSG 指令触发后执行的命令操作几乎是一样的，比如说对模块的复位操作或模块的重组态操作。

第 15 章

Add-On 指令的创建和编程

Add-On 指令直译为附加指令，意为附加于控制器指令系统的指令，其实际意义是用户自定义指令，用户可以自己创建并命名指令，编辑相应的执行代码和建立相应的数据结构，满足个性化或标准化的程序处理。

Add_On 指令将编辑特定的程序执行代码，控制器运行时通过后台调用执行，其程序执行代码的编辑支持梯形图、功能块和语句结构三种编程语言，可选择其中之一的编程语言来编写惟一的逻辑代码例程，或诸如预扫描、后扫描和梯级条件为假的特殊处理的扫描模式例程。

与此产生的还有与 Add-On 指令同名的预定义结构数据类型，也即指令后台运行所涉及的数据。指令数据分为本地标签和指令参数两种类别，内部使用的本地标签可满足逻辑功能运用或存放自定义指令处理数据的中间结果，其数据封闭性可防止意外修改并具有保密性。预先定义的指令参数用来完成程序处理所需的对外数据交换，可直接查看或设置。有的数据还可设定显现在指令界面上，这样的指令参数可直接键入数据并提供指令界面的数据实时监视。请注意，每条 Add-On 指令运行的数据是自动隔离的，并拥有自己独立的结构数据标签，每条指令将指定专用的标签存在于数据库中，其数据结构类型为同名 Add-On 指令的预定义结构数据类型。

编制完成后的 Add-On 指令出现在控制器的指令系统中，它的运用跟控制器系统的标准指令几乎没有区别，Add-On 指令支持控制器的梯形图、功能块、顺序功能流程图（SFC）和语句结构四种编程语言的调用。此外，已创建并存在于指令系统的 Add-On 指令还可以被后面创建的 Add-On 指令引用，这种嵌套式的引用，最多可达 7 层。

Add-On 指令可以用来完成个性化处理，用户在项目中需要反复执行的逻辑代码不必编写子例程的调用，将其编制成 Add-On 指令的调用比子例程要方便得多，并拥有专用数据。对于特定的具有保密性质的程序处理，Add-On 指令可以封装，形成黑匣子功能，以保护开发者的专利。有别于传统产品的项目整体封存保密，由于指令的独立封装，可以为最终用户开放整个项目，以方便系统的在线维护和管理。

Add-On 指令可以用来建立标准化处理，将行业或企业拟定的标准编制成 Add-On 指令，发布给使用部门，Add-On 指令不但可以在同一项目的例程中反复引用，还可以在不同项目的例程中使用。Add-On 指令可以从创建它的项目中导出，以独立的文本文件形式存放，并可将该文件导入任何一个项目，加入任何一个项目的指令系统中，供例程中的编程调用。

15.1 Add-On 指令的创建过程和引用

下面我们通过一个实例来看 Add-On 指令是如何创建和组态的，并且通过梯形图梯级的指令引用来观察它的运行情况。

这里，我们采用一个熟悉的例子。前面曾经编写过的保持最新数据的堆栈数组处理的梯级逻辑，现在我们把这一段程序处理装到 Add-On 指令中来运用。先分析一下案例。

Add-On 指令将对外交换的数据有：

- 数据装入源地址：提供堆栈数组的数据来源，DINT 数据类型标签；
- 数据卸载目标地址：提供堆栈数组的数据目标，DINT 数据类型标签；
- 数据采集周期时间：提供堆栈数组的数据采集周期，DINT 数据类型标签；
- 堆栈数组指针清零：提供对堆栈数组的位置管理，BOOL 数据类型标签；
- 堆栈数组：传递堆栈数组的数据，20 个 DINT 数据类型的数组。

指令执行要求：

- 将源地址的数据装入堆栈数组；
- 堆栈数组的最前面的数据卸载到目标地址；
- 在 Add-On 指令参数中输入数据采集的时间和堆栈数组。

在 RSLogix5000 编程软件的树形结构中，选中“Add-On Instructions”项，右击出现如图 15-1 所示的画面。

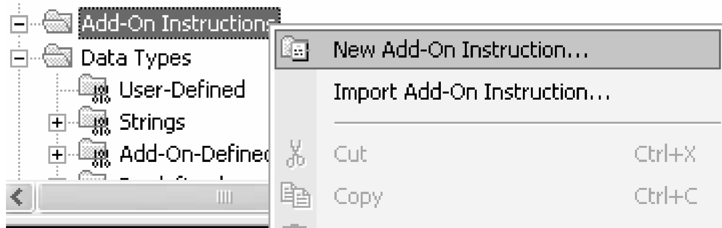


图 15-1　选择创建新的 Add-On 指令

点击“New Add-On Instruction”项，进入创建页面，如图 15-2 所示。

创建一个 Add-On 指令，本页包含的信息说明：

- Name：为 Add_On 指令命名，此名称不但是指令的名称，指令所属的自定义结构数据，也沿用相同名称。
- Description：对指令加以说明，作为注释信息，可使用汉字。
- Type：执行代码的例程类型，可以是梯形图、功能块或语句结构编程。
- Revision：指令版本，可设定主要版本、次要版本和延伸文字。
- Vendor：指令卖主名称。

为该 Add-On 指令命名 STACK，点击 OK 后完成创建，在编程软件的树形结构中出现了新的 Add_On 指令，如图 15-3 所示。创建时选择了梯形图，所以 Logic 编程将采用梯形图。

双击“STACK”，打开指令定义页面，如图 15-4 所示。

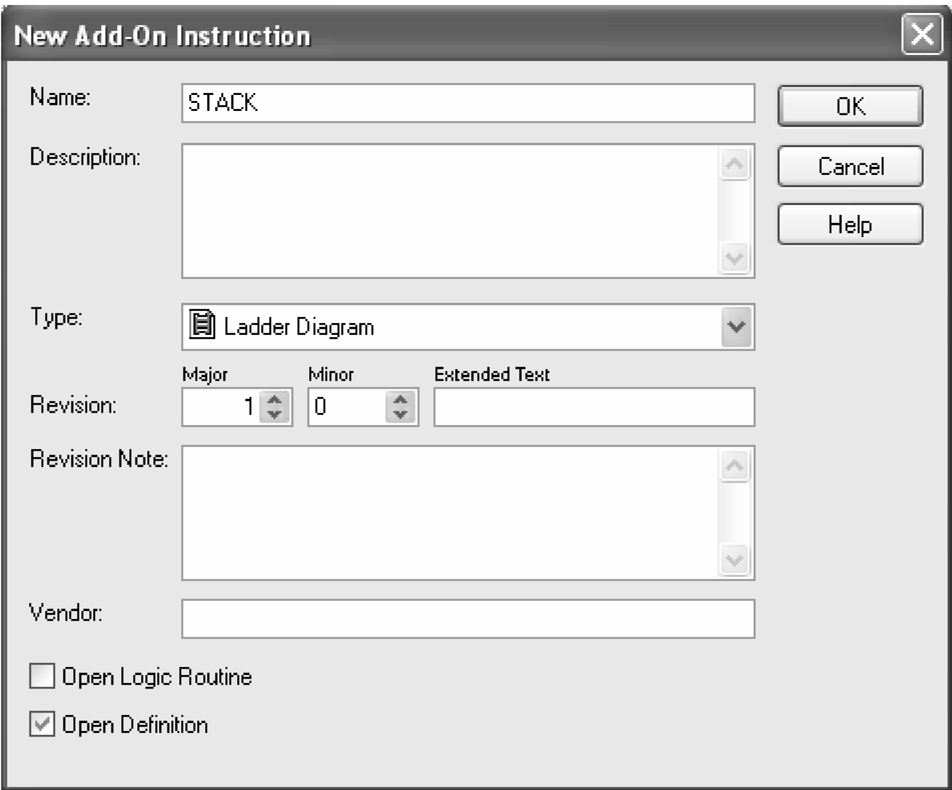


图 15-2　Add-On 指令创建页面

General 页面跟创建时的画面差不多，点击左下角的 Logic ，即可进入编程，此例是梯形图编程。左下角还显示了本指令结构数据所耗用的字节，也是每次引用指令指定数据标签所占用的字节数。Add-On 指令的结构数据由下面要讲到的参数和本地标签构成。

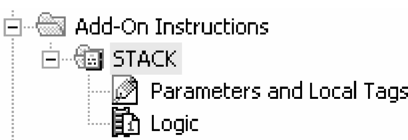


图 15-3　产生新的 Add-On 指令

点击 Parmeters 页面，进入参数组态，创建指令所需要的参数，如图 15-5 所示。

参数说明：

- Name：为 Add_On 指令中的参数命名。
- Usage：参数的交换方式选择，有三种用法：
  - Input（输入参数）：将参数数据带入指令后台，只能是基本数据，即操作数。
  - Output（输出参数）：从指令后台带出参数数据，只能是基本数据，即操作数。
  - InOut（输入/输出参数）：指令后台与外部的批量数据交换，可以是基本数据，也可以是结构数据和数组。这部分不占用指令结构数据的耗用字节，编辑指令引用时键入的数据结构必须跟这里定义的数据结构是一样的。
- Data Type：参数的数据类型，和一般定义标签相同。
- Default：参数的默认值，指令编程引用时，在创建指令结构数据标签时给出的初始值。

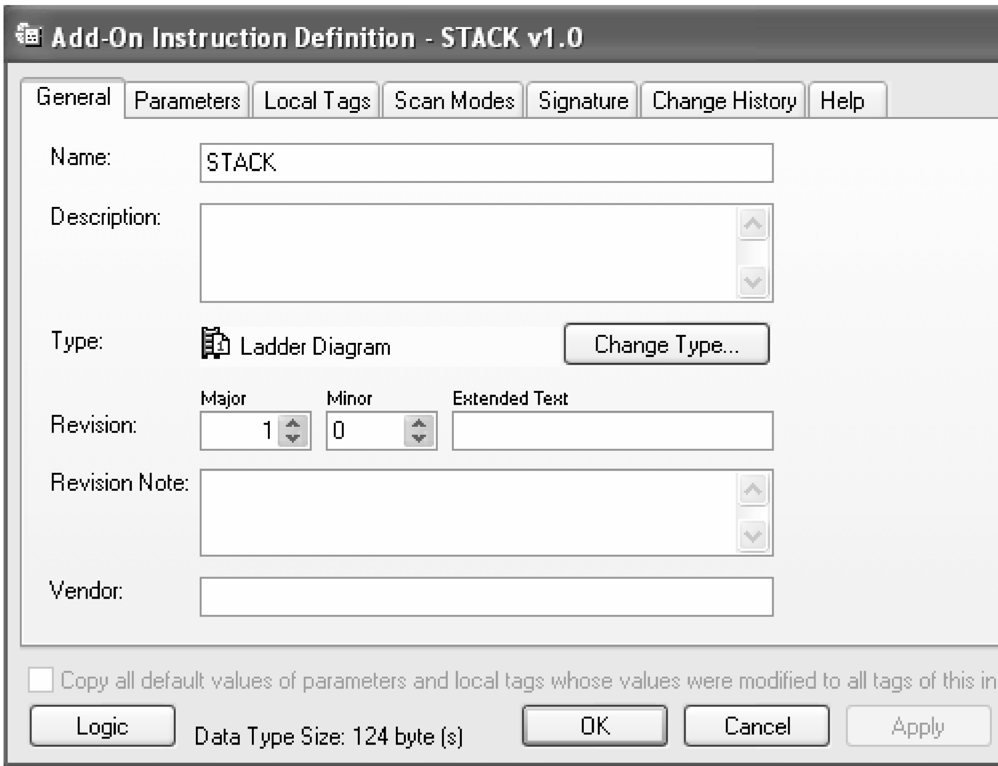
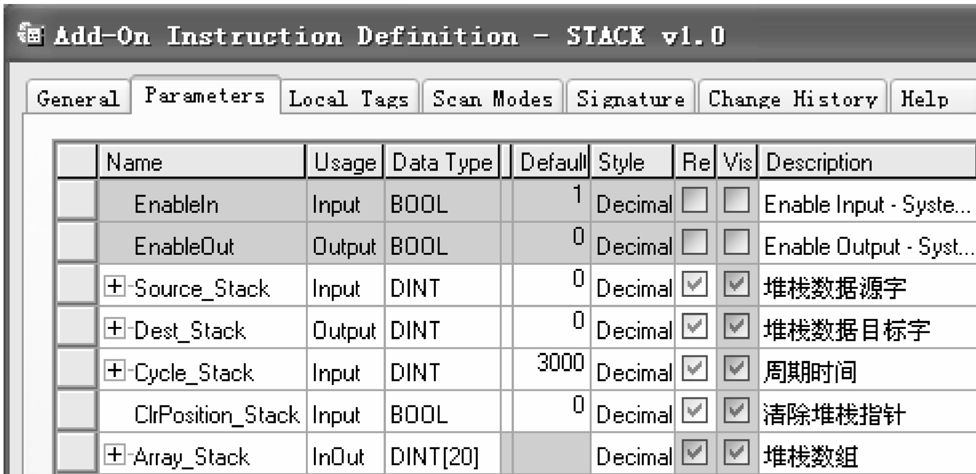


图 15-4　Add-On 指令定义页面



Name	Usage	Data Type	Default	Style	Re	Vis	Description
EnableIn	Input	BOOL	1	Decimal	☐	☐	Enable Input - Syste...
EnableOut	Output	BOOL	0	Decimal	☐	☐	Enable Output - Syst...
+Source_Stack	Input	DINT	0	Decimal	☑	☑	堆栈数据源字
+Dest_Stack	Output	DINT	0	Decimal	☑	☑	堆栈数据目标字
+Cycle_Stack	Input	DINT	3000	Decimal	☑	☑	周期时间
ClrPosition_Stack	Input	BOOL	0	Decimal	☑	☑	清除堆栈指针
+Array_Stack	InOut	DINT[20]		Decimal	☑	☑	堆栈数组

图 15-5　Add-On 指令参数组态页面

- Style：参数数据的表达形式，和一般定义标签相同。
- Req：此项勾选，该参数不但出现在指令中，并实施指令的后台数据对外部数据进行 COP 式的交换。
- Vis：此项勾选，参数将出现在指令中，如果单独勾选此项而没有选择 Req，则在指令上表现为只读数据。

总而言之，Add_On 指令参数是指令对外数据交换的接口，通过输入参数将数据传递进去，交给指令后台运行，然后输出参数将运行结果传递出来。根据本例的要求，一共建立了 5 个对外数据交换的参数，并按照用途，选定了数据类型。参数表格中已一一列举说明。

源字 Source_Stack 采用 Input 数据类型，该数据类型只允许基本数据类型，即操作数，这里选用的是一个双整字。目标字 Dest_Stack 采用 Output 数据类型，该数据类型只允许基本数据类型，即操作数，这里选用的是一个双整字。

周期时间 Cycle_Stack 选用 Input 数据类型，这里选用的是一个双整字。有一点要提请注意的是周期时间的单位，这显然跟 Logic 中要编写的梯级逻辑的处理有关，我们将使用计时器来完成周期性动作，计时器的时间基值是 1ms，作为计时器预置值的时间提供，这里也应该为毫秒的时间单位。当默认值为 3000 时，意味着默认的数据采集时间是每 3 秒一次。清除堆栈指针操作位 ClrPosition_Stack 选用的 Input 数据类型作为梯级条件的操作数，这里选用的是布尔量。

堆栈数组 Array_Stack 采用的是 InOut 数据类型，该数据类型允许结构数据或数组进行交换。这里指定的堆栈数组是 20 个双整字的数组，也即实际交换的堆栈数组长度，而并非堆栈指令操作的数组长度。正如我们在第 10 章的讨论，先入先出堆栈指令 FFL 和 FFU 操作的数组要比实际数组多出一个数据单元，我们把这个问题交给 Add_On 指令内部来处理，在外部保持跟使用的长度一致。

点击 Local Tags 页面，进入本地标签组态，如图 15-6 所示。

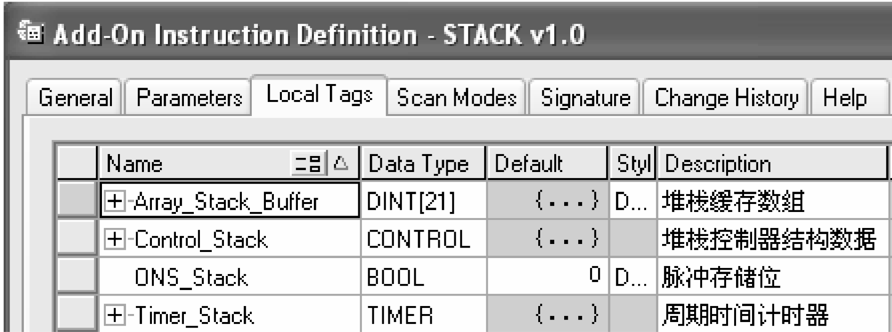


图 15-6　Add-On 指令本地标签组态页面

本地标签是完全运行在 Add-On 指令后台的数据，也是 Add-On 指令后台运行的中间结果，这种隐藏，既保护了这些参数不被修改，也给封装后的 Add-On 指令提供了数据保护。

用于堆栈指令操作的堆栈数组 Array_Stack_Buffer，设定的数组长度为 21，比外部交换的堆栈数组多一个数组元素，它的前 20 个数据将复制给指令参数的堆栈数组。

控制结构标签 Control_Stack 是堆栈指令操作所使用的控制结构数据标签，其指针将在合适的时候被复位为 0。

窄脉冲存储位 ONS_Stack 为确保指针复位操作只执行一次的 ONS 指令而设。

计时器 Timer_Stack 用于设定采样周期时间，其定时工作的完成位 DN 为堆栈指令提供了执行条件。

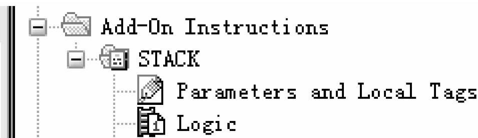


图 15-7　Add-On 指令的逻辑例程

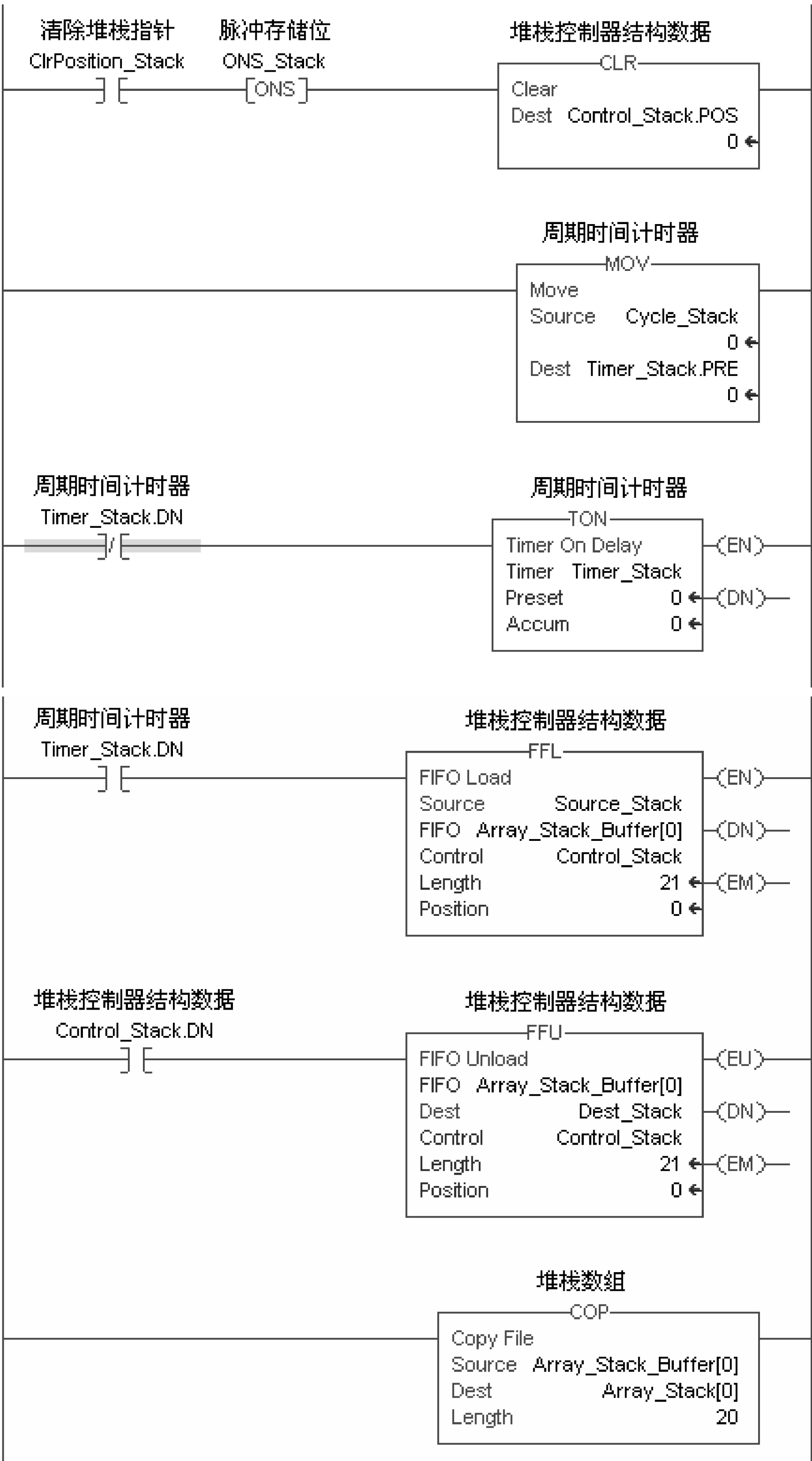


图 15-8 堆栈处理的梯级逻辑

现在，我们为自定义指令的执行，编写梯级逻辑，如图 15-7 所示，梯形图 Logic 就是存放 Add-On 指令执行逻辑的地方，我们所说的自定义指令后台运行就是在执行 Add-On 指令里面的逻辑。

点击 Logic进入梯形图编写梯级逻辑，如图 15-8 所示。

传送指令 MOV 将外部设定的周期采样时间传送到计时器预置值，计时器完成位在这个设定的时间到达时置位，产生一个触发脉冲，堆栈装载指令 FFL 执行，将堆栈源地址 Source_Stack 的数据装入堆栈缓存数组 Array_Stack_Buffer，直到堆栈满，即 Control_Stackz. DN 位置位，堆栈卸载指令 FFU 执行，将堆栈缓存数组最前面的数据卸出到堆栈目标地址 Dest_Stack，腾出最后一个数据单元，堆栈装载指令 FFL 再装入新的数据。

最后一个梯级将堆栈缓存数组 Array_Stack_Buffer 的前 20 个数据复制给堆栈数组 Array_Stack，并通过指令交换到外部。这样，使用 STACK 指令丝毫不用考虑堆栈数组的长度问题，需要多长的数组，直接使用多长的数组便可。

前次程序的运行，堆栈数组 Add-On 指令的堆栈指针也许不在零的位置，这种残留的状态很有可能影响到当前的执行，Add-On 指令应该提供一个可以清除堆栈指针的位指令操作。显然，这种操作只能在必要时进行一次性操作，为了外部清除条件更为宽松，操作不受限制，我们可以把限制一次性操作的 ONS 指令编写在 Add-On 指令的后台逻辑操作。

至此，一个 Add-On 指令就基本建成了，此时在指令集的 Add-On 分类项中出现了新的指令 STACK，如图 15-9 所示。这个指令不但能在梯形图中引用，功能块和语句结构编程照样也可以引用。



图 15-9　新的指令 STACK

同时在系统结构数据中，也出现了与 STACK 指令同名的 STACK 预定义结构数据，如图 15-10 所示。就像系统的标准指令一样，这是每当 STACK 指令被引用时，分配给指令的结构数据标签所必用的结构数据类型。

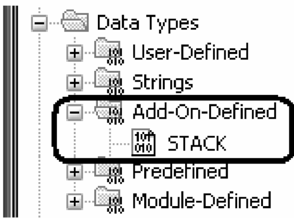


图 15-10　STACK 预定义结构数据

现在，我们在一个普通的梯形图例程中编写引用这条指令的梯级逻辑，如图 15-11 所示。

这里，我们将模拟量通道 0 的数据当作源堆栈参数。一般来说，外部模拟量的输入也可以直接引用在指令的 Source_Stackde 参数项中，但我们定义的 Source_Stackd 是双整字。在此例中，模拟量模块的输入是整型数数据类型 INT，不能被 Source_Stackd 参数项直接接受，所以通过 MOV 指令转换一下，MOV 指令允许目标参数项和源参数项使用不同的数据类型。当然，采用定标指令代替 MOV 指令也未尝不可，下面我们就要谈到定标指令。

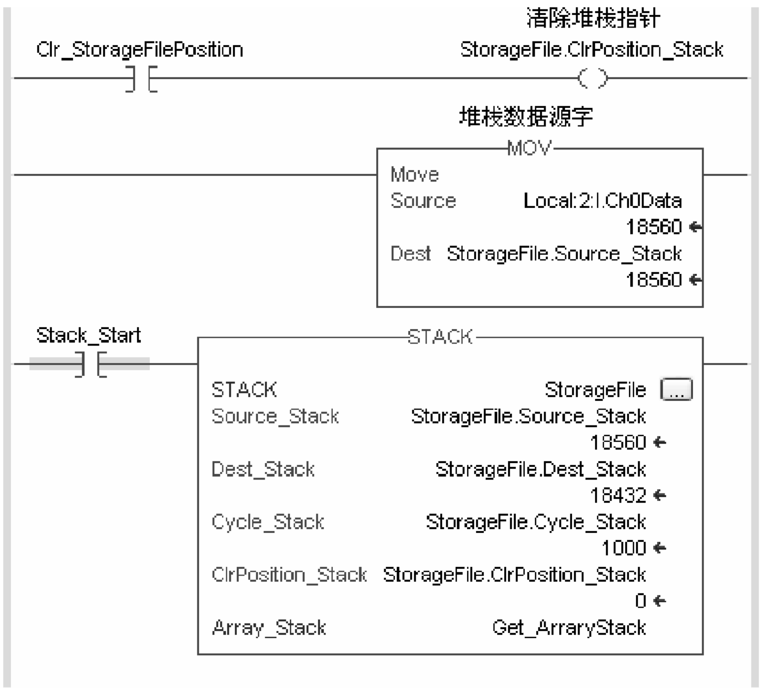


图 15-11　在例程中引用 STACK 指令的梯级逻辑

堆栈数组 Array_Stack 项键入的地址必须跟指令定义设定的 InOut 参数结构完全一致，这里我们设定的是 20 个长度的双整字数组。当在线查看数据时，我们可以看到如图 15-12 所示的数据显示。

Name	Value
− Get_ArraryStack	{...}
+ Get_ArraryStack[0]	18176
+ Get_ArraryStack[1]	18176
+ Get_ArraryStack[2]	18304
+ Get_ArraryStack[3]	18304
+ Get_ArraryStack[4]	18176
+ Get_ArraryStack[5]	18304
+ Get_ArraryStack[6]	18432
+ Get_ArraryStack[7]	18688
+ Get_ArraryStack[8]	18688
+ Get_ArraryStack[9]	18304
+ Get_ArraryStack[10]	18432
+ Get_ArraryStack[11]	18560
+ Get_ArraryStack[12]	18688
+ Get_ArraryStack[13]	18432
+ Get_ArraryStack[14]	18432
+ Get_ArraryStack[15]	18304
+ Get_ArraryStack[16]	18432
+ Get_ArraryStack[17]	18688
+ Get_ArraryStack[18]	18560
+ Get_ArraryStack[19]	18432

图 15-12　STACK 指令的堆栈数据显示

刚才，我们编写了堆栈指针从外部复位的做法，当外部操作复位时，内部有响应，这为用户根据运用情况进行复位提供了机会，如果我们规定 STACK 指令在梯级条件不成立的时候一定要复位堆栈指针，不妨利用扫描模式的介入来实现梯级条件不成立时自动复位堆栈指针。

扫描模式是 Logic 执行之外的某些处理，用于执行的特殊介入，进入 Scan Modes 页面，如图 15-13 所示的画面，这是定义预扫描、后扫描以及梯级条件为假时所进行的相关处理，按照事先编辑的梯级逻辑执行。

点击“EnableInFalse routine”的 New... ，进入如图 15-14 所示的画面，这里将编写梯级条件不成立时执行的代码，同样的，可以是梯形图、功能块和语句结构编程来完成执行代码。

点击“OK”，完成了梯级条件未使能的处理逻

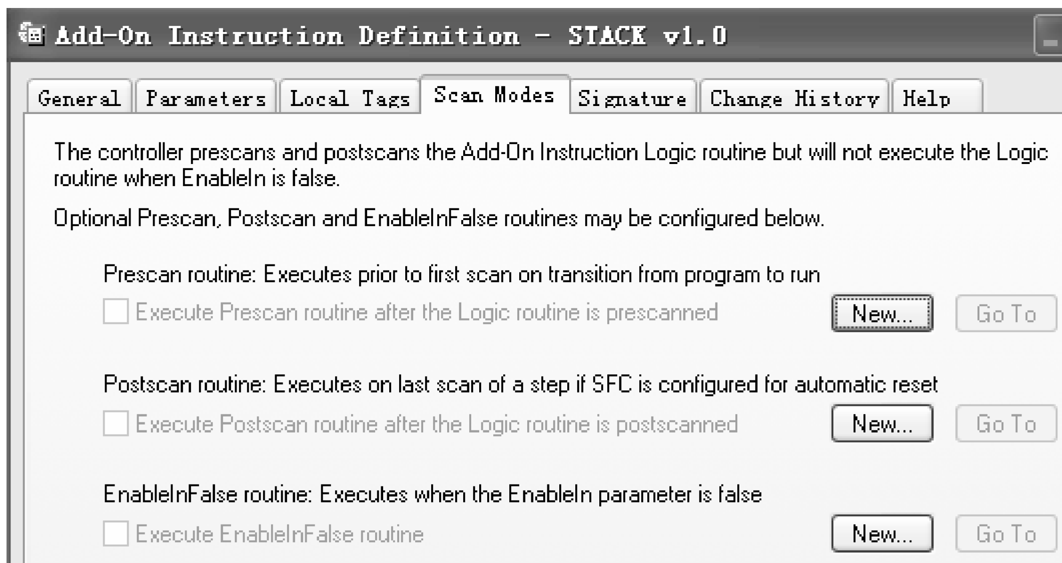


图 15-13　扫描模式页面

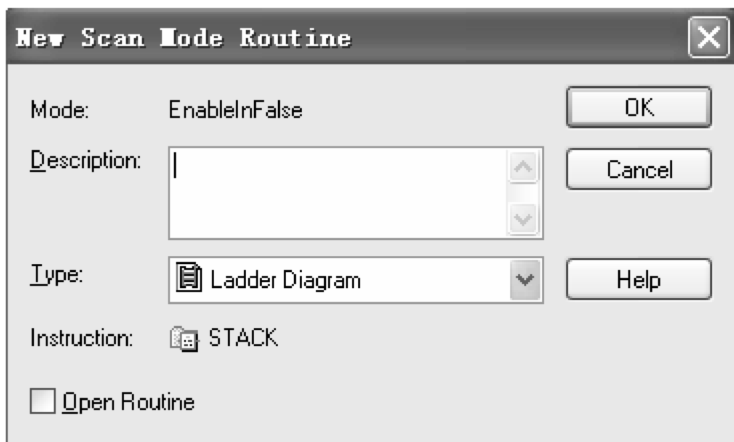


图 15-14　创造 Add-On 指令不使能执行的例程

辑例程的创建，如图 15-15 所示。

由此，一个梯级条件未使能的逻辑例程产生，如图 15-16 所示，用于编写梯级条件不成立时的逻辑处理，大多数情况下会编写 Add-On 指令初始化处理的执行代码。

点击 Go To ，或点击 EnableInFalse，进入 EnableInFalse 的例程，可在该例程中编写清除堆栈指针的梯级逻辑，如图 15-17 所示。

这样，堆栈指针在 STACK 指令梯级条件给定的调用和停止调用中，不知不觉就被清除了。要知道堆栈数组的处理，指针可是一个关键，严谨的指针处理可以避免操作上的混乱。当然关于 STACK 指令的使用指南就得说明，每当指令梯级条件使能，堆栈数组将从 0 元素开始装载，直到堆栈数组装满才开始推陈出新。

如果创建预扫描例程并在其中编写同样的逻辑，例程运行之初，同样可以获得这样的清

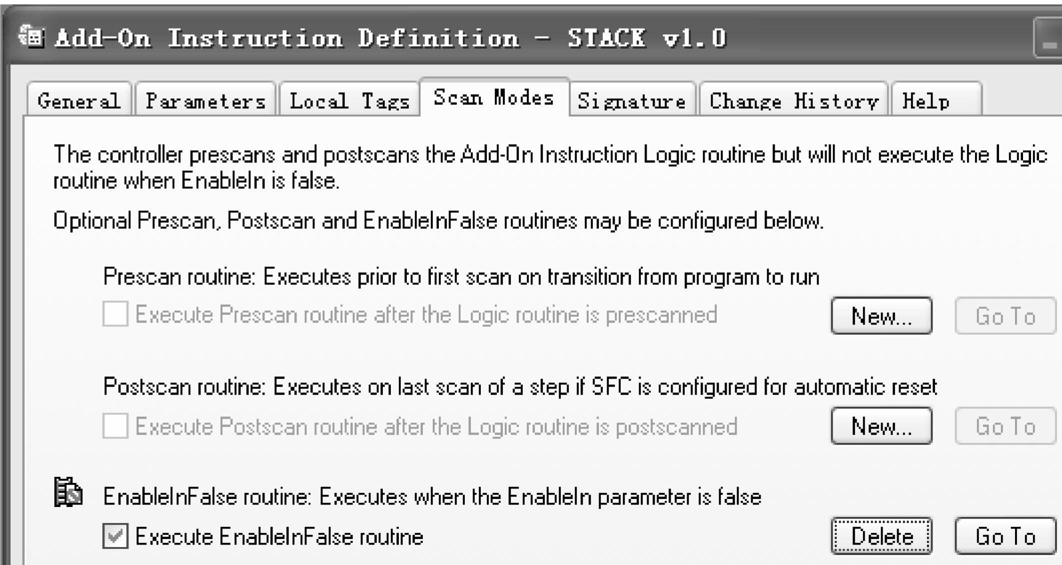


图 15-15　扫描模式页面

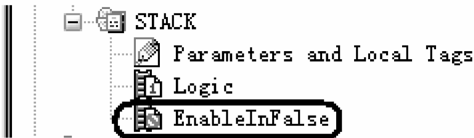


图 15-16　一个梯级条件未使能的逻辑例程出现



图 15-17　清除堆栈指针的梯级逻辑

理，这可以避免当例程首次扫描，本条指令的梯级条件正处在 ON 状态而引起的问题。

当我们修改了 Add_On 指令的逻辑执行动作时，之前所有已编写并使用的指令、逻辑关系均随之发生了变化，例如堆栈指针在梯级条件不成立时堆栈指针复位的有关处理。

要想修改参数设定的默认值，情况却不是那么简单，比如我们修改了采样周期的默认时间，从 3000 改为 2000，然后我们又编制了新的 STACK 指令运用，你会惊奇的发现，在修改之前编写的 STACK 指令并不遵从修改后的默认值，依旧使用原来的 3000 的默认值，只有修

改后编写的 STACK 指令才会使用新的默认值 2000。这是一个值得注意的问题，因为编写 STACK 指令时创建相应的 Add_On 指令的结构数据标签按照当时的默认值已经设定，修改 Add_On 指令参数时，无法同时改变已经创建的 Add_On 指令结构数据标签中的默认值，如果你希望所有的指令都使用新的默认值，可以到已经创建的 Add_On 指令结构数据标签中逐个地进行修改。

当然，每个默认值在指令运行时会自动给出，大多数情形下默认值为 0，因而不被人们所关注。尽管每个参数一旦运行之后给出的默认值是可以更改的，但是重新运行又会给出默认值。本例中周期时间的默认值是 3000，运行后可以从默认值更改为任何数据，但是重新运行又会给出默认值 3000。

还有 3 个页面需要简单介绍。

如图 15-18 所示的签名页面是 Safety Add_On 指令专用的，签名是安全产品特定的做法，我们在这里不作讨论。

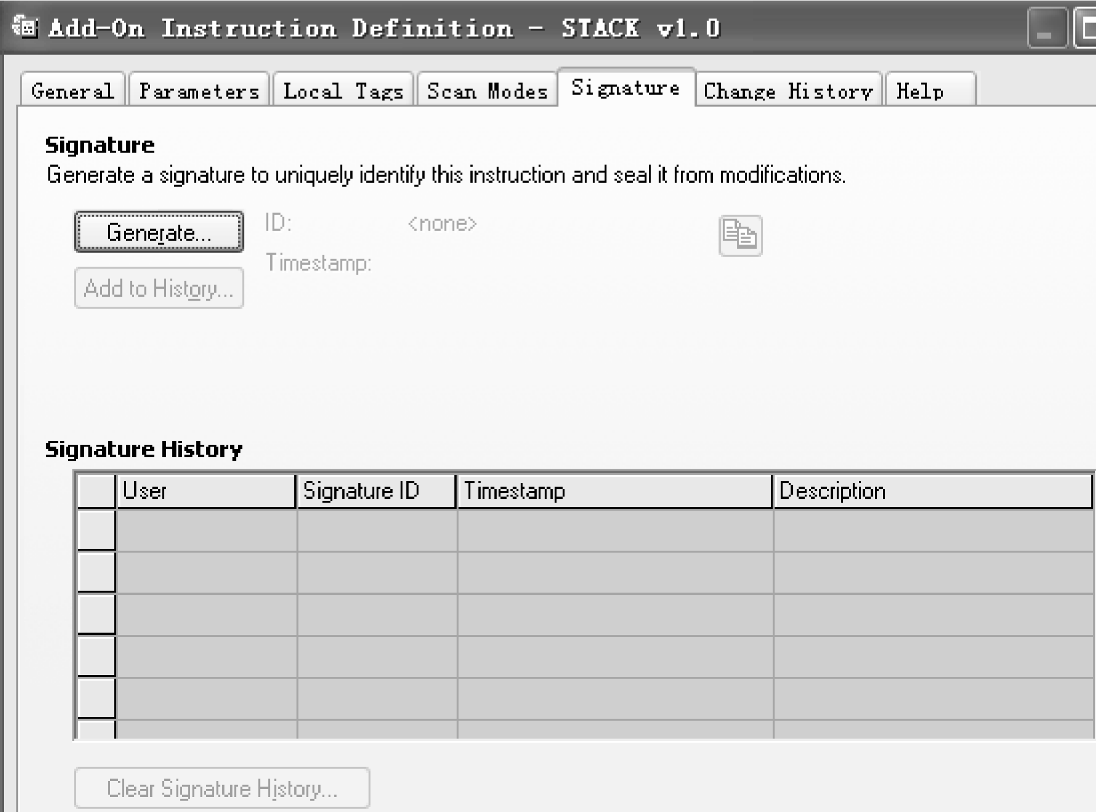


图 15-18　安全签名页面

点击“Change History”，进入如图 15-19 所示画面。这个页面用来记录指令创建和历次修改的情况，如果需要，也可以予以清除。

点击“Help”，进入帮助画面，如图 15-20 所示，这里将出现帮助文件，这些文字说明也是 Add_On 指令的组成部分，为了让使用者更好更正确地使用指令，在此提供更为详尽的信息。

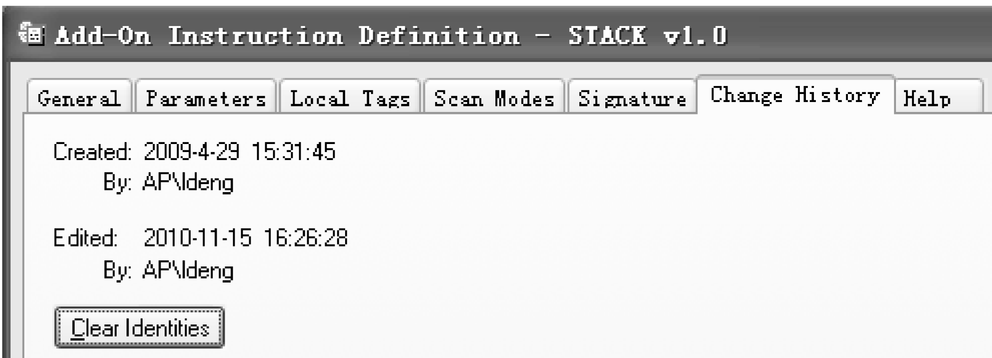


图 15-19　记录指令的创建和修改的页面

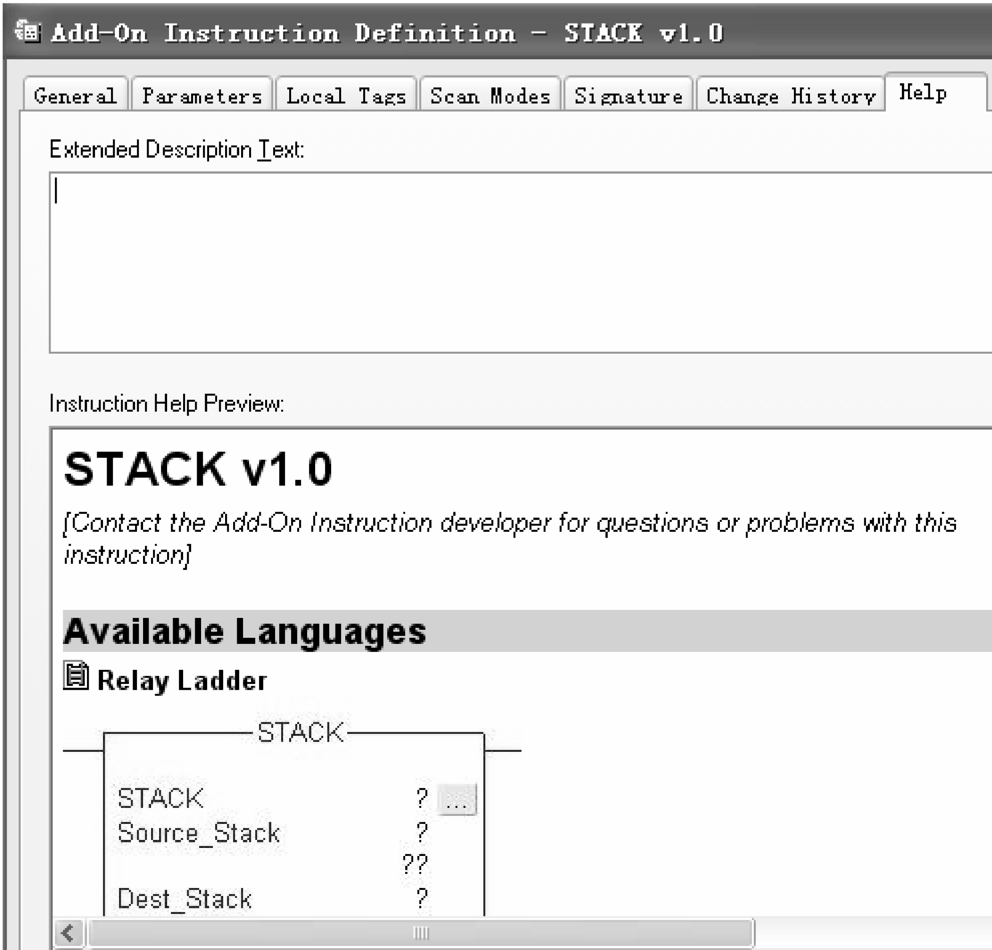


图 15-20　帮助页面

15.2　将常规动作转化为 Add-On 指令的实例

对重复使用的执行动作，运用 Add_On 指令来完成也是十分成功的。下面我们再尝试建一条 Add-On 指令，指令的要求是：当这条指令调用时，有一个输出点闪烁，其闪烁频率可更改。我们同样可以借用前面编写过的关于闪烁的实例来作为 Add-On 指令的逻辑处理。

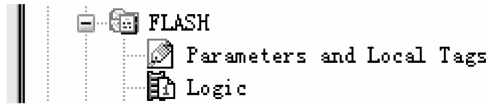


图 15-21　创建一个名为 FLASH 的 Add-On 指令

再创建一个名为 FLASH 的 Add-On 指令，如图 15-21 所示。

创建过程是相似的，这里略去。

进入 Parameters 页面设置外部参数，如图 15-22 所示。外部参数建立了闪烁频率的双整字输入参数以及闪烁的布尔量输出参数，都勾选在外部显示。

Add-On Instruction Definition - FLASH v1.0

General | Parameters | Local Tags | Scan Modes | Signature | Change History | Help

Name	Usage	Data T	Al	Defau	Style	Req	Vis	Description
EnableIn	Input	BOOL		1	Decimal	☐	☐	Enable Input -
EnableOut	Output	BOOL		0	Decimal	☐	☐	Enable Output
+ Frequency_Flash	Input	DINT		400	Decimal	☑	☑	闪烁频率
Out_Flash	Output	BOOL		0	Decimal	☑	☑	闪烁输出点

图 15-22　在 Parameters 页面设置外部参数

进入 Local Tags 页面设置本地标签，如图 15-23 所示，本地标签即为指令的内部数据，多为指令执行的中间结果，在指令执行时，受到保护不会被修改。本例中是一个计时器结构数据，这个计时器控制了时间，即闪烁的频率，当这个计时器的预置值被修改时，输出点的闪烁频率也被修改。预置值的默认值为 400。

进入 Logic 例程编写梯级逻辑，令指令的输出点完成闪烁的功能，如图 15-24 所示。

除了跟我们在第 7 章比较指令的运用中编写的实例一样，令输出点闪烁之外，还添加了一个 MOV 指令的梯级传递外部的数据，跟随外部更改的闪烁频率参数。

进入普通的例程进行编写，可以看到指令集的 Add-On 类别项中又增添了新的成员（FLASH 指令），如图 15-25 所示。

在普通例程中引用的 FLASH 指令，如图 15-26 所示。

让程序运行起来，测试 FLASH 指令执行的过程，发现有问题。当梯级条件给定，灯将按照预定的频率闪烁，并可以修改闪烁的频率；当梯级条件不成立时，输出却停留在不确定的状态。

改变梯级条件，令其成立或不成立，反复测试几次，发现停止 FLASH 指令执行之后，有时灯是亮着的，有时灯是熄灭的。显然，当梯级条件消失的时候，取决于当时计时器的累加值处于什么阶段，闪烁的输出会不确定在 1 或是 0 的状态。

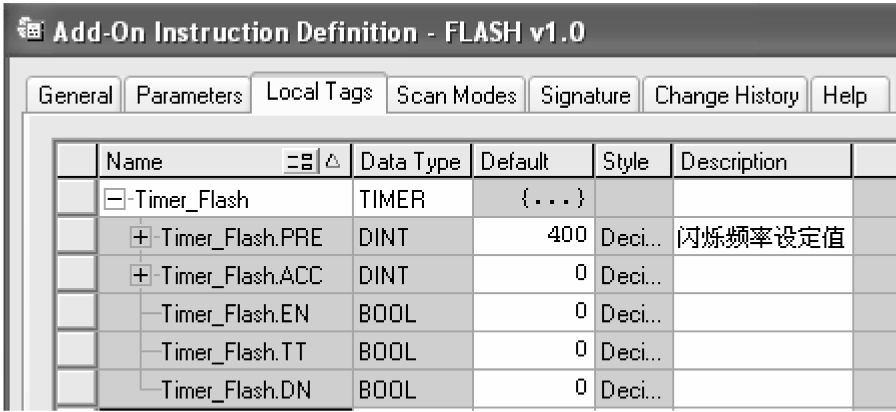


图 15-23　在 Local Tags 页面设置本地标签

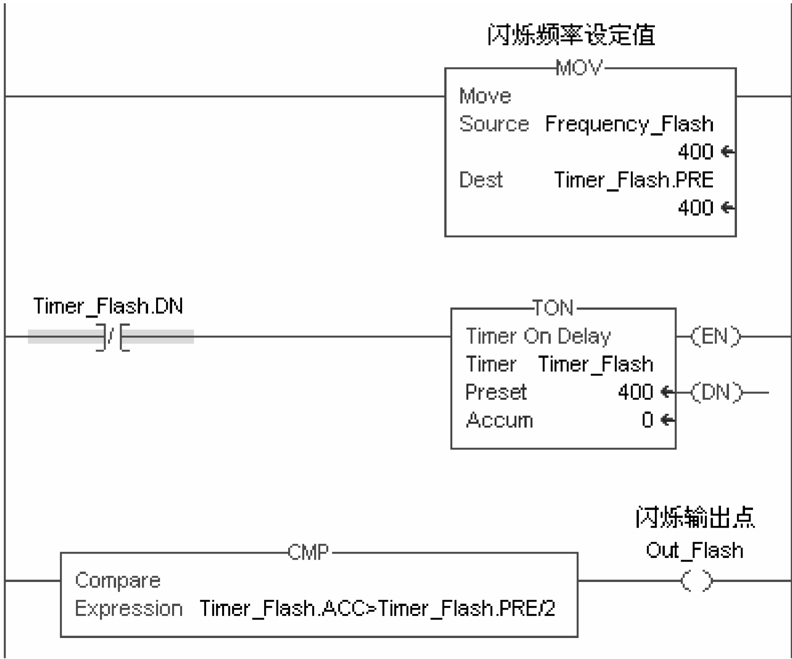


图 15-24　闪烁点的梯级逻辑



图 15-25　Add-On 类别项中增添了 FLASH 指令

现在，我们来修改一下，令梯级条件不成立时固定在复位状态 0。

进入 Scan Modes 页面，如图 15-27 所示的画面，这是定义预扫描、后扫描以及梯级条件为假时所进行的相关处理，将按照事先编辑的逻辑执行。

点击“EnableInFalse routine”的 New...，进入如图 15-28 所示的画面，这里将指定梯

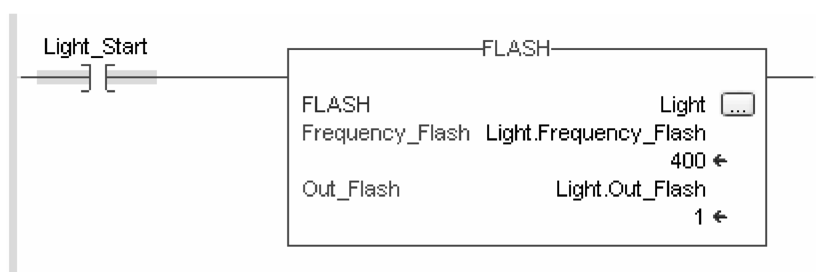


图 15-26　在例程中编写调用 FLASH 指令的梯级逻辑

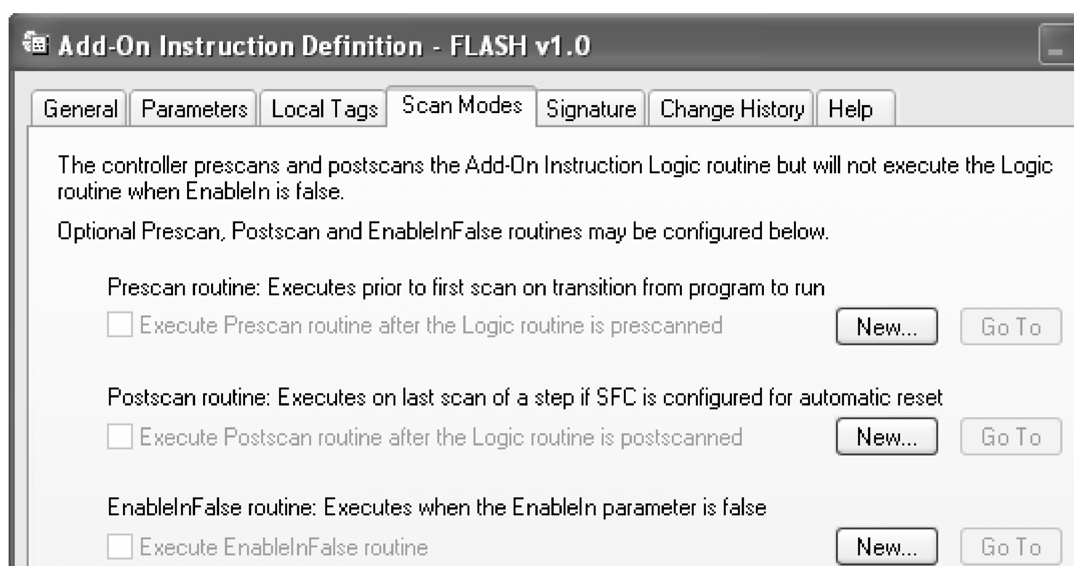


图 15-27　Scan Modes 页面

级条件不成立时执行的逻辑。同样的，可以是梯形图、功能块和语句结构编程。

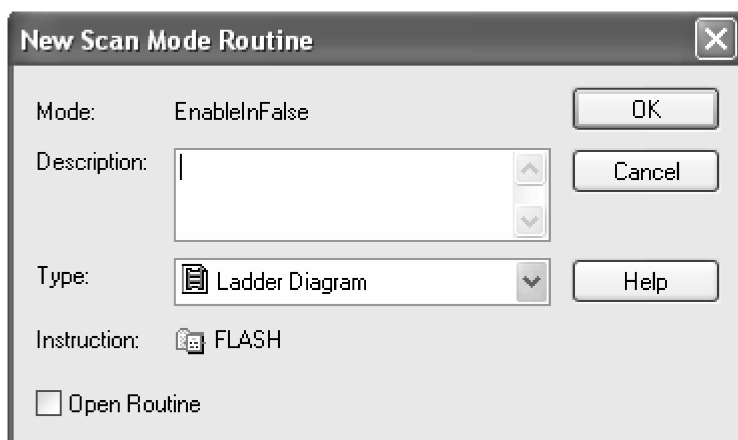


图 15-28　创建梯级条件不使能的执行例程

点击 OK ，产生了梯级条件使能为假时应该执行的例程的定义画面，如图 15-29 所示。

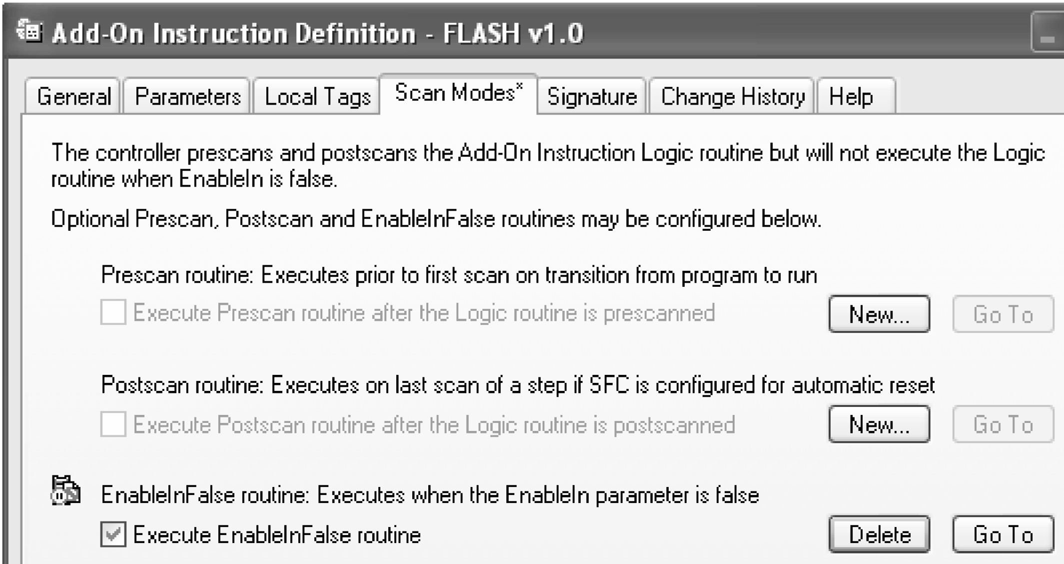


图 15-29　梯级条件使能为假时应该执行的例程的定义画面

与此同时，FLASH 指令也增加了新的结构 EnableInFalse，如图 15-30 所示。

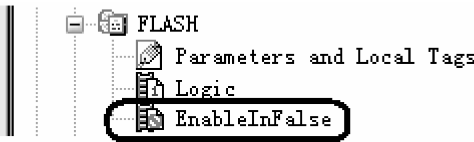


图15-30　FLASH 指令增加梯级不使能执行例程

点击 Logic，或点击 EnableInFalse，进入在 EnableInFalse 的例程，并在例程中编写梯级逻辑，如图 15-31 所示。

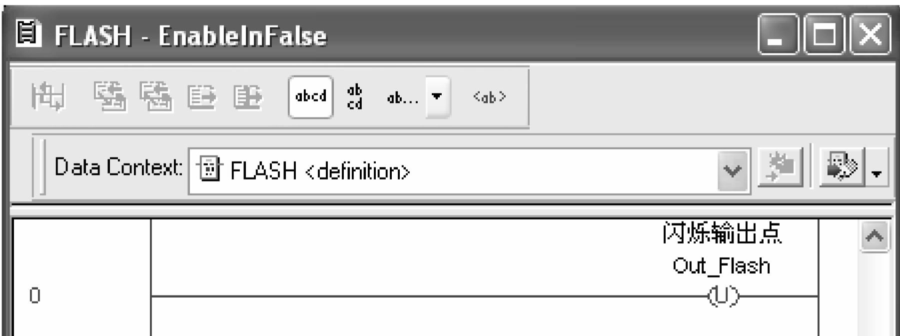


图 15-31　复位闪烁点的梯级逻辑

这是在梯级条件不成立的时候，EnableInFalse 例程所要执行的动作。它将解锁这个位，无论指令停止调用时它处在什么样的状态，最后总会令这个位复位为零。

运行程序，检查 FLASH 指令执行结果，问题得到了解决。

重复的动作由某些难以理解的梯级逻辑组成，一个完整的独立功能均可以采用用户自定义指令解决。下面，我们再尝试将产生正弦波的梯级逻辑整合为正弦波指令。首先，创建产生正弦波的名为 SINE_WAVE 的 Add_On 指令，如图 15-32 所示。

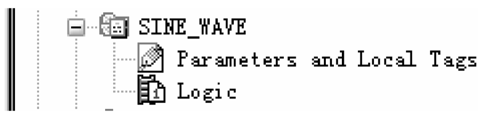


图 15-32　创建 SINE_WAVE 指令

创建指令参数，如图 15-33 所示，输入参数 Period 带入周期时间，单位为秒，默认值为 5；输入参数 Max 作为正弦波的幅值，默认值为 10；输出参数则带出正弦波变化值。

Add-On Instruction Definition - SINE_WAVE v1.0

General | Parameters | Local Tags | Scan Modes | Signature | Change History | Help

Name	Usage	Data Type	Alias	Defau	Style	Req	Vis	Description
EnableIn	Input	BOOL		1	Decimal	☐	☐	Enable Inpu...
EnableOut	Output	BOOL		0	Decimal	☐	☐	Enable Out...
+-Period	Input	DINT		5	Decimal	☑	☑	周期
Max	Input	REAL		10.0	Float	☑	☑	幅值
Out_SineWave	Output	REAL		0.0	Float	☑	☑	波形输出值

图 15-33　创建指令参数

创建本地标签，如图 15-34 所示，这是正弦波计算过程所需要用到的几个中间结果数据，这些数据无需对外。

Add-On Instruction Definition - SINE_WAVE v1.0

General | Parameters | Local Tags | Scan Modes | Signature | Change History | Help

Name	Data Type	Default	Styl	Description
+-Timer_Sine	TIMER	{...}		
+-PI	DINT	0	D...	
+-XNum	DINT	0	D...	
+-XDen	DINT	0	D...	

图 15-34　创建本地标签

像第 6 章算术逻辑运算指令的编程中关于正弦波的运算一样，编辑有关的梯级逻辑，如图 15-35 所示，周期时间秒的单位通过乘法指令 MUL 换算成毫秒的单位。

在例程中编写指令调用，如图 15-36 所示，当梯级条件成立时，连绵不断的正弦波输出，通过标签 Sine_Wave1. Out_SineWave 送出。

这个指令所产生的正弦波可以提供一个模拟信号，通过改变周期值的大小来调试模拟硬件电路的高通滤波回路或低通滤波回路。

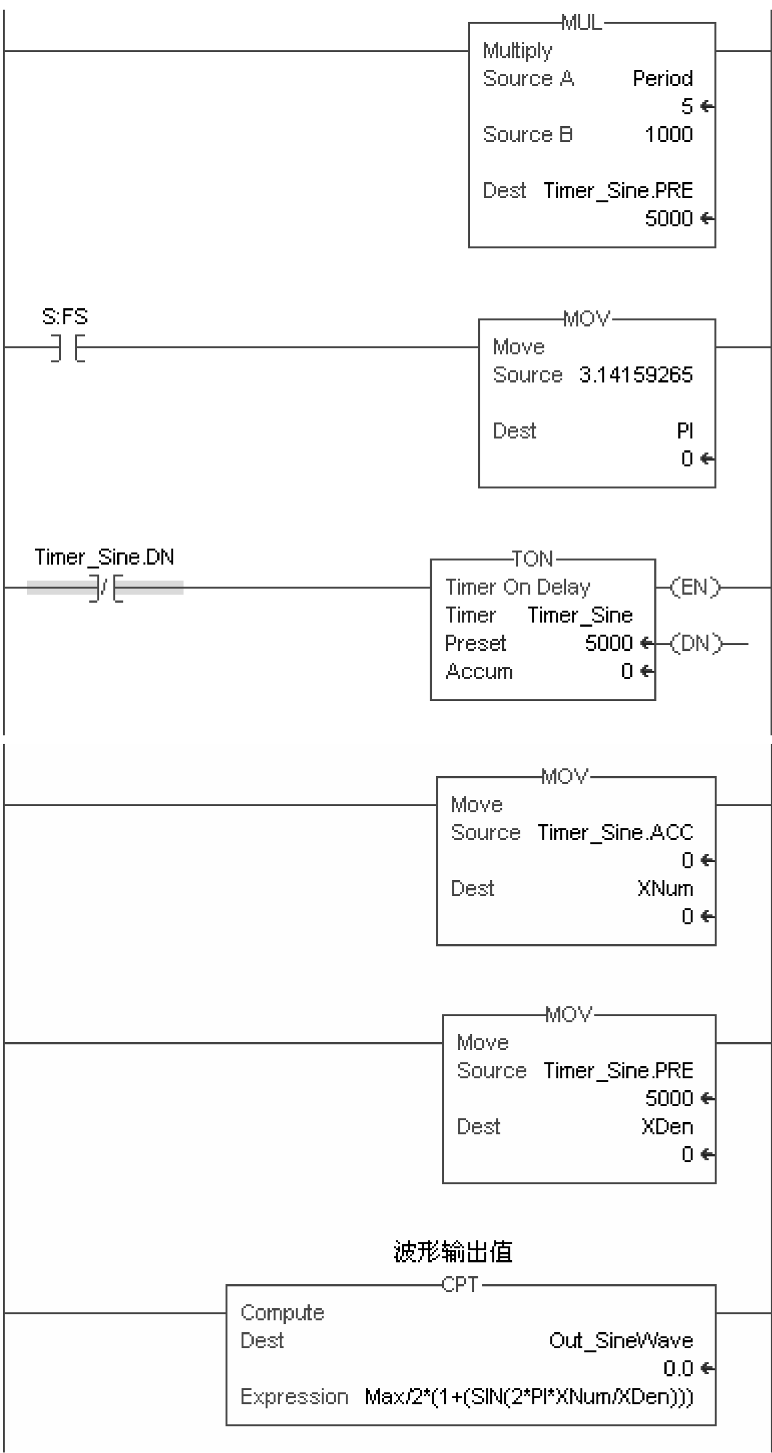


图 15-35　正弦波产生的梯级逻辑

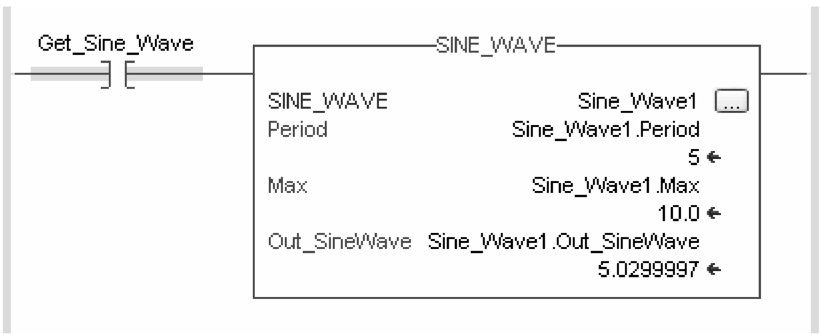


图 15-36　在例程中编写调用 SINE_WAVE 指令的梯级逻辑

15.3　将信息处理过程转化为 Add-On 指令的实例

我们再来看一个实例处理，记录事件发生系统日期时间的一个数组，需要转换为 ASCII 数组送到外部 ASCII 设备，编程进行转换的大量重复操作。首先创建 Add_On 指令 GET_ASCII_DATE_TIME，如图 15-37 所示。

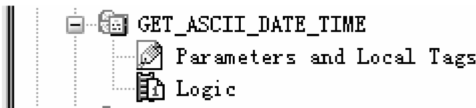


图 15-37　创建 Add-On 指令 GET_ASCII_DATE_TIME

根据需求，创建将要带入 GET_ASCII_DATE_TIME 指令的记录系统日期时间数组，此数组为用户自定义数据结构的数组，转换后带出的 ASCII 码数组为字符串数组，如图 15-38 所示。

Add-On Instruction Definition - GET_ASCII_DATE_TIME v1.0

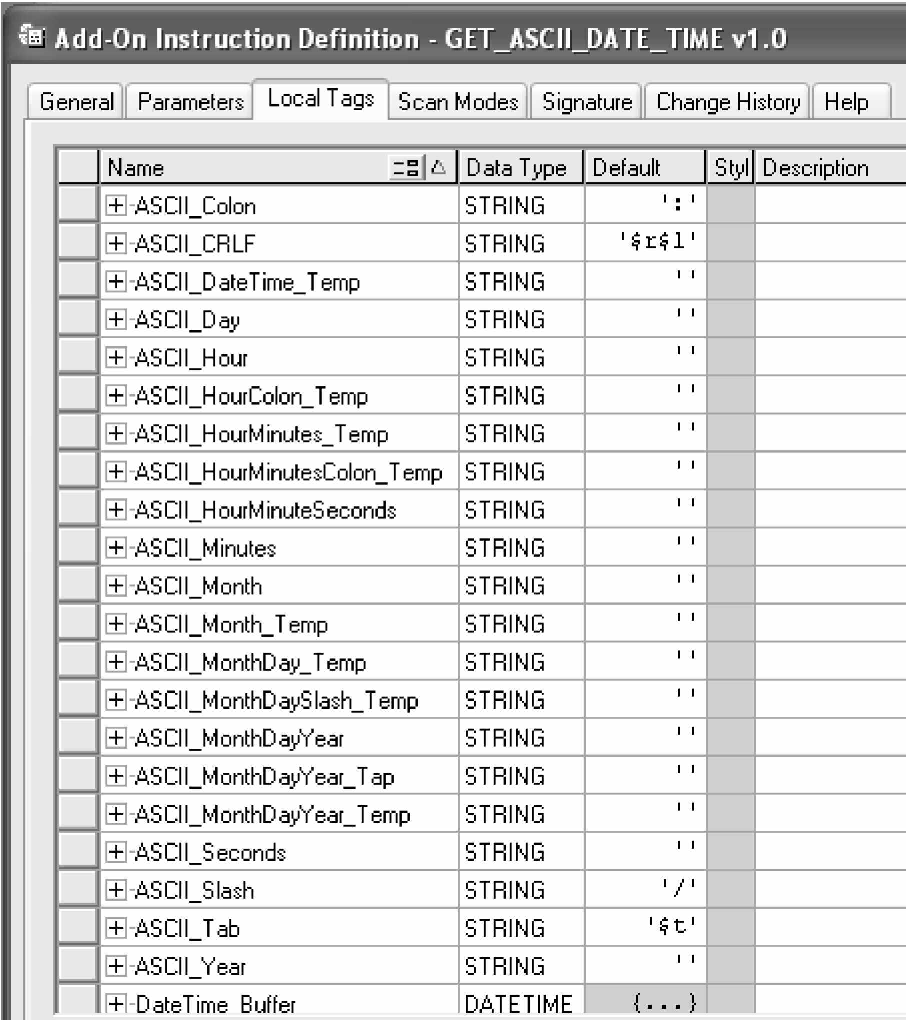
General　Parameters　Local Tags　Scan Modes　Signature　Change History　Help

Name	Usage	Data Type	Al	Defau	Style	Req	Vis
EnableIn	Input	BOOL		1	Decimal	☐	☐
EnableOut	Output	BOOL		0	Decimal	☐	☐
⊞ Record_DateTime	InOut	DATETIME				☑	☑
⊞ ASCII_DateTime	InOut	STRING				☑	☑
						☐	☐

图 15-38　创建指令参数

根据需求，创建本地标签，这些均为梯级逻辑运行所需要的中间变量，和我们之前在第 8 章传送和转换指令的编程中讨论过的类似，并在 ASCII_Colon、ASCII_CRLF、ASCII_Slash、ASCII_Tab 4 个标签中设置相应的默认值，即某些符号或后缀，如图 15-39 所示。

在 Logic 例程中编写相关的转换处理梯级逻辑，如图 15-40 所示，首先将记录日期时间



Name	Data Type	Default	Styl	Description
ASCII_Colon	STRING	':'		
ASCII_CRLF	STRING	'rl'		
ASCII_DateTime_Temp	STRING	''		
ASCII_Day	STRING	''		
ASCII_Hour	STRING	''		
ASCII_HourColon_Temp	STRING	''		
ASCII_HourMinutes_Temp	STRING	''		
ASCII_HourMinutesColon_Temp	STRING	''		
ASCII_HourMinuteSeconds	STRING	''		
ASCII_Minutes	STRING	''		
ASCII_Month	STRING	''		
ASCII_Month_Temp	STRING	''		
ASCII_MonthDay_Temp	STRING	''		
ASCII_MonthDaySlash_Temp	STRING	''		
ASCII_MonthDayYear	STRING	''		
ASCII_MonthDayYear_Tap	STRING	''		
ASCII_MonthDayYear_Temp	STRING	''		
ASCII_Seconds	STRING	''		
ASCII_Slash	STRING	'/'		
ASCII_Tab	STRING	'$t'		
ASCII_Year	STRING	''		
DateTime_Buffer	DATETIME	{...}		

图 15-39　创建本地标签

Record_DateTime 复制至缓冲单元 DateTime_Buffer，分别对日期的年、月、日和时间的时、分、秒进行相应的转换，获得对应的 6 个 ASCII 码。将日期连接起来，再将时间连接起来，拼接为 ASCII_DateTime 字符串作为最后的输出。

GET_ASCII_DATE_TIME 指令每次只能转换一条记录，要编写循环执行的梯级逻辑来操作。在运行的例程中编写梯级逻辑如图 15-41 所示，当 Get_ASCII 为 0，设定的梯级条件成立，Go_On 跳转到后面梯级，不执行 GET_ASCII_DATE_TIME 指令；当 Get_ASCII 为 1，设定的梯级条件不成立，进入 GET_ASCII_DATE_TIME 指令的跳转循环执行，直到 50 个元素转换完毕，获得了转换的结果。

用跳转指令来完成这个 50 个元素的循环操作，当进入循环时，将指针 Get_ASCII_Index 清除为 0，完成 GET_ASCII_DATE_TIME 指令对数组 0 元素的操作，这一次操作后，指针

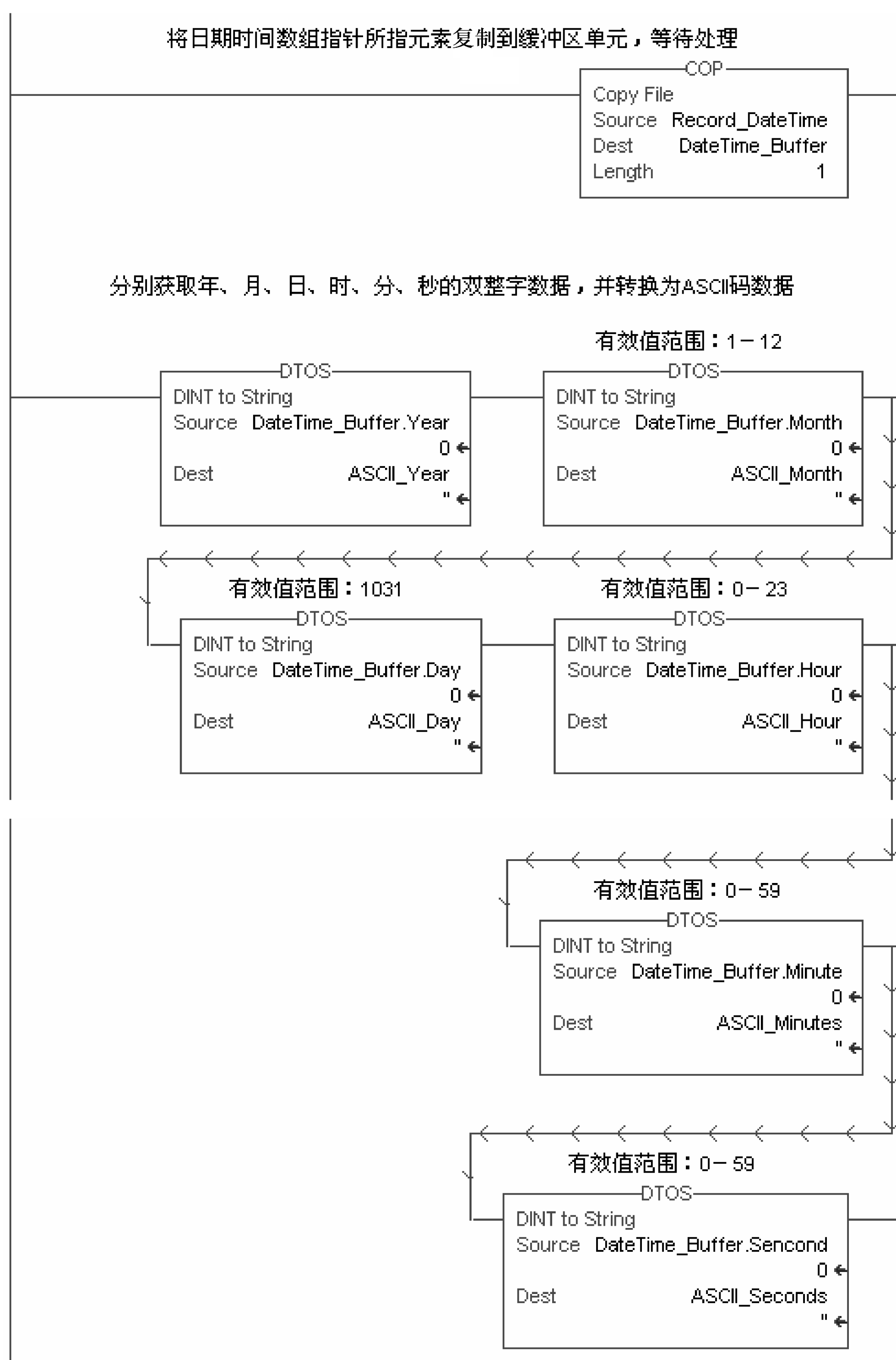


图 15-40　日期时间转换处理的梯级逻辑

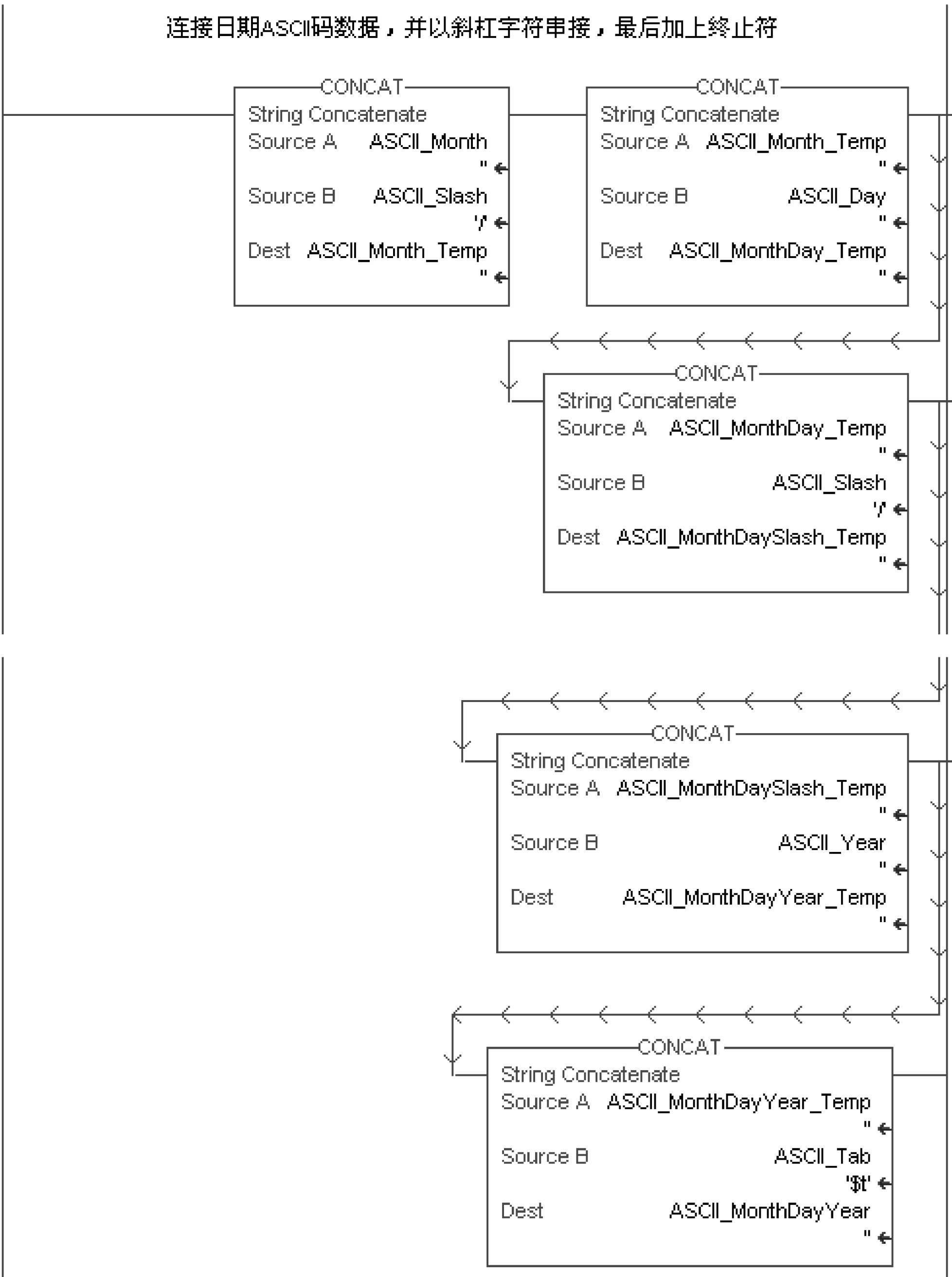


图 15-40 日期时间转换处理的梯级逻辑（续）

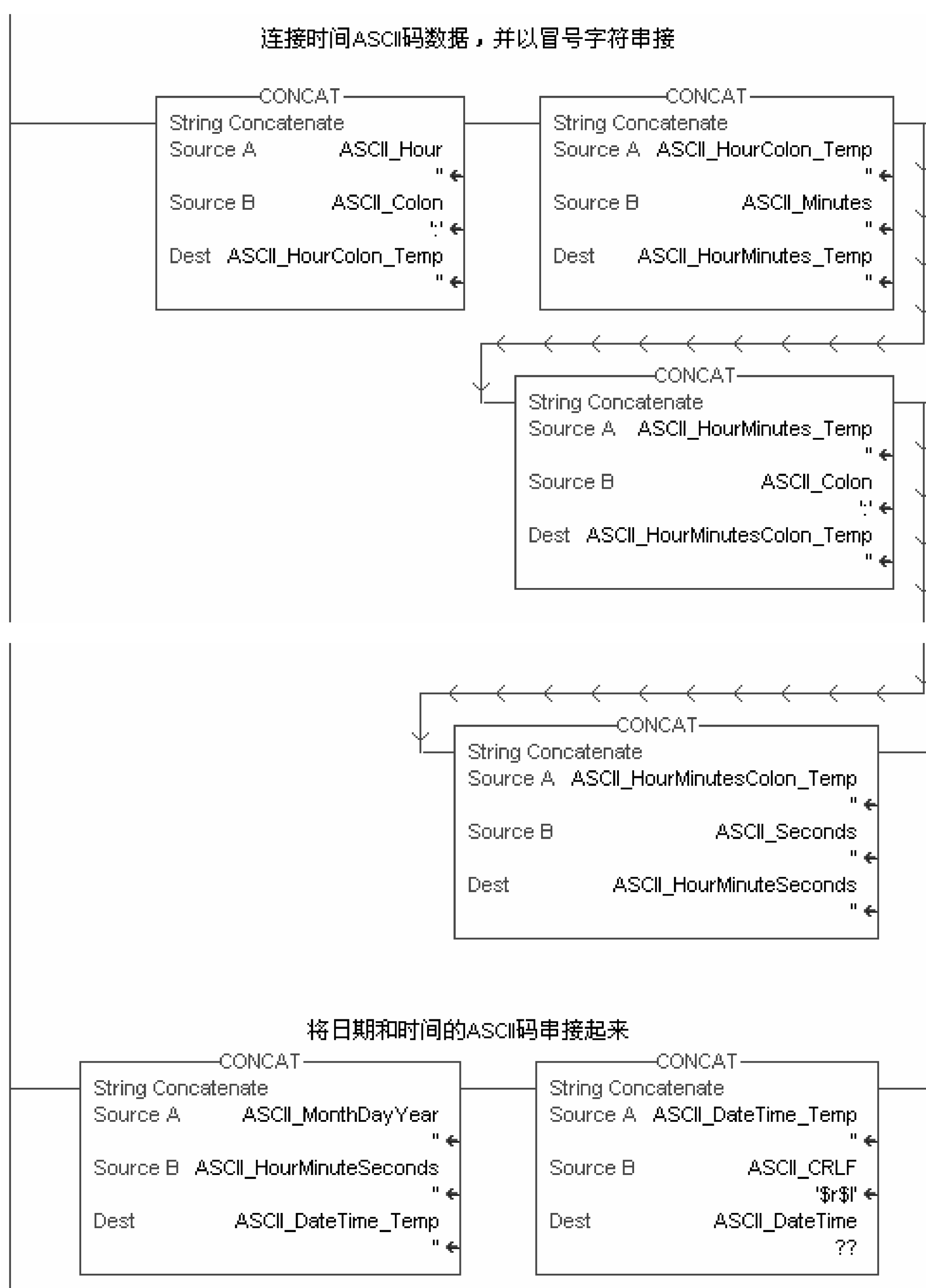


图 15-40　日期时间转换处理的梯级逻辑（续）

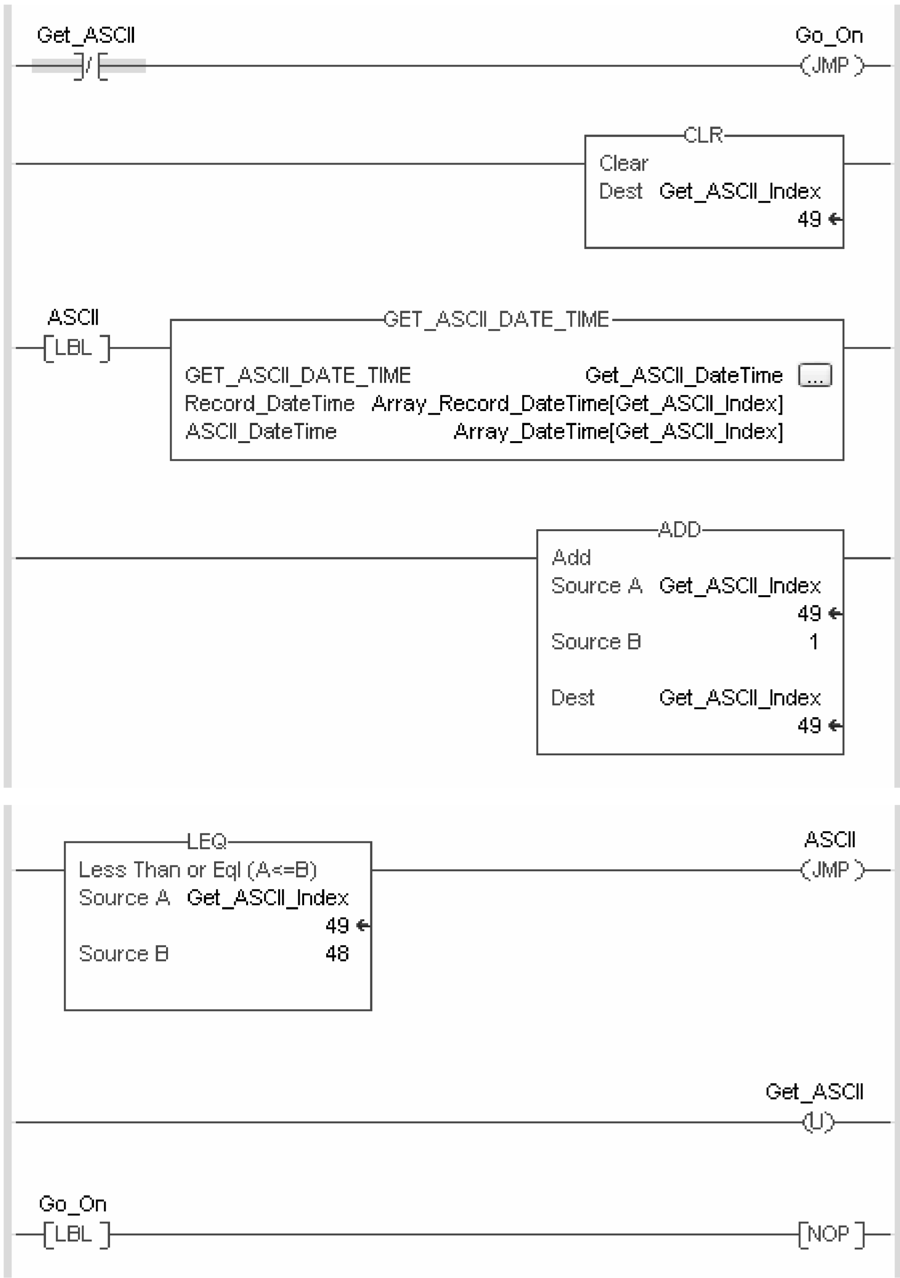


图 15-41　在例程中编写调用 GET_ASCⅡ_DATA_TIME 指令的梯级逻辑

加 1，经过比较指令 LEQ 的判定，跳回 GET_ASCII_DATE_TIME 指令所在梯级继续执行，直到判断指针等于 48 时，仍然梯级条件成立，跳回 GET_ASCII_DATE_TIME 指令所在梯级执行第 49 元素的操作，指针加 1 后，判断条件不成立，离开循环操作，由此，从 0 ~ 49 的 50 个元素全部转换完毕。转换的一轮操作完成之后，解锁 Get_ASCII，避免再次进入循环，确保每次循环操作获得的是同期数据。

GET_ASCII_DATE_TIME 指令带入的参数和带出的参数均采用间接寻址，通过 Get_ASCII_Index 标签的指针作用来修改数组寻址地址。根据 Add_On 指令的执行规则，Array_Record_DateTime［Get_ASCII_Index］复制给 GET_ASCII_DATE_TIME 指令的后台数据 Record_DateTime；GET_ASCII_DATE_TIME 指令的后台数据 ASCII_DateTime 复制给 Array_ASCIIDateTime［Get_ASCII_Index］。

将编写好的测试例程下载到控制器，运行例程后可以查看一下转换的结果，分别截取 Array_Record_DateTime 数组和 Array_ASCIIDateTime 数组最前面的两个数据和最后面的两个数据观察，如图 15-42 和图 15-43 所示。

− Array_DateTime	{...}
+ Array_DateTime[0]	'6/12/2011 $t14:55:41$r$l'
+ Array_DateTime[1]	'6/12/2011 $t14:55:46$r$l'

− Array_Record_DateTime	{...}
− Array_Record_DateTime[0]	{...}
+ Array_Record_DateTime[0].Year	2011
+ Array_Record_DateTime[0].Month	6
+ Array_Record_DateTime[0].Day	12
+ Array_Record_DateTime[0].Hour	14
+ Array_Record_DateTime[0].Minute	58
+ Array_Record_DateTime[0].Sencond	51
+ Array_Record_DateTime[0].Micros...	986624
− Array_Record_DateTime[1]	{...}
+ Array_Record_DateTime[1].Year	2011
+ Array_Record_DateTime[1].Month	6
+ Array_Record_DateTime[1].Day	12
+ Array_Record_DateTime[1].Hour	14
+ Array_Record_DateTime[1].Minute	58
+ Array_Record_DateTime[1].Sencond	56
+ Array_Record_DateTime[1].Micros...	990902

图 15-42　查看最前面的两个数据

+ Array_DateTime[48]	'6/12/2011 $t14:59:42$r$l'
+ Array_DateTime[49]	'6/12/2011 $t14:59:47$r$l'

− Array_Record_DateTime[48]	{...}
+ Array_Record_DateTime[48].Year	2011
+ Array_Record_DateTime[48].Month	6
+ Array_Record_DateTime[48].Day	12
+ Array_Record_DateTime[48].Hour	15
+ Array_Record_DateTime[48].Minute	3
+ Array_Record_DateTime[48].Senc...	23
+ Array_Record_DateTime[48].Micro...	892478
− Array_Record_DateTime[49]	{...}
+ Array_Record_DateTime[49].Year	2011
+ Array_Record_DateTime[49].Month	6
+ Array_Record_DateTime[49].Day	12
+ Array_Record_DateTime[49].Hour	15
+ Array_Record_DateTime[49].Minute	3
+ Array_Record_DateTime[49].Senc...	28
+ Array_Record_DateTime[49].Micro...	897772

图 15-43　查看最后面的两个数据

15.4　将标准规则转化为 Add-On 指令的实例

行业惯有的规则或长期的实践经验往往会形成某些固有的解决方案，最终演变为行业标准或企业规则。这些标准或规则固然有文字和管理的约束，但也延伸和嵌入到了控制系统中，尤其是一些技术方面的处理。

通过 Add_On 指令的推行来实现一个企业或一个行业的标准，按照规定对某事物或对象进行标准化的处理。下面我们针对一个标准化的诊断处理编写一条 Add_On 指令。

这是一个具有监视模拟量输入功能的指令，针对模拟量数据监视，如图 15-44 所示。

要求当来自设备检测的模拟量反馈值超出上限或下限值的时间持续在 n 秒之后，指令报警，报警信息确认之后，给予报警复位。标准化编程进行延时报警处理，报警的上、下限值和延时时间均可以由编程者或操作者来设置。

创建一个名为 ANALOG 的 Add_On 指令，如图 15-45 所示。

创建如图 15-46 所示的参数设置，各个指令参数的用法和数据类型以及说明如表中所示，并设定参数显现在 Add-On 指令中。这里监视时间设定值的单位是秒，这意味着在 Logic 例程中编写梯级逻辑时将有特殊的处理。众所周知，计时器的计时基值是毫秒，对使用者来说，如果没有精度的特别要求，总是习惯使用秒为时间单位。

创建如图 15-47 所示的本地标签，高限值和低限值的计算中间结果存放单元，高限监视

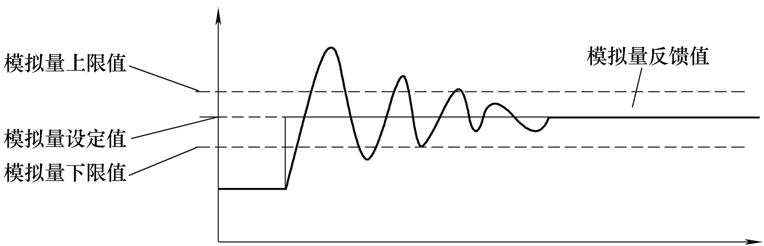


图 15-44　针对模拟量数据进行监视

延时计时器和低限监视延时计时器。

编写梯级逻辑如图 15-48 所示。用加法指令 ADD 和减法指令 SUB 分别计算出高限比较值和低限比较值。延时计时器的预置值要进行换算，指令设置时间单位规定为秒，计时器时间

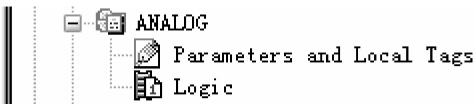


图 15-45　创建一个名为 ANALOG 的 Add_On 指令

Add-On Instruction Definition - ANALOG v1.0

General | Parameters | Local Tags | Scan Modes | Signature | Change History | Help

Name	Usage	Data Type	Al	Default	Style	Req	Vis	Description
EnableIn	Input	BOOL		1	Decimal	☐	☐	Enable Input ...
EnableOut	Output	BOOL		0	Decimal	☐	☐	Enable Outpu...
Analog_Set	Input	REAL		0.0	Float	☑	☑	设定值
High_Tolerance	Input	REAL		0.0	Float	☑	☑	高限偏差
Low_Tolerance	Input	REAL		0.0	Float	☑	☑	低限偏差
+ Monitor_Timer_Sec	Input	DINT		5	Decimal	☑	☑	监视时间设定
Analog_Test	Input	REAL		0.0	Float	☑	☑	监视值
High_Alarm	Output	BOOL		0	Decimal	☑	☑	高限报警
Low_Alarm	Output	BOOL		0	Decimal	☑	☑	低限报警
Ack_Alarm	Input	BOOL		0	Decimal	☑	☑	报警确认

图 15-46　创建指令参数

Add-On Instruction Definition - ANALOG v1.0

General | Parameters | Local Tags | Scan Modes | Signature | Change History | Help

Name	Data Type	Default	Style	Description
High_Limit	REAL	0.0	Float	上限值
Low_Limit	REAL	0.0	Float	下限值
+ MonitorHigh_Timer	TIMER	{...}		高报警监视计时器
+ MonitorLow_Timer	TIMER	{...}		低报警监视计时器

图 15-47　创建指令标签

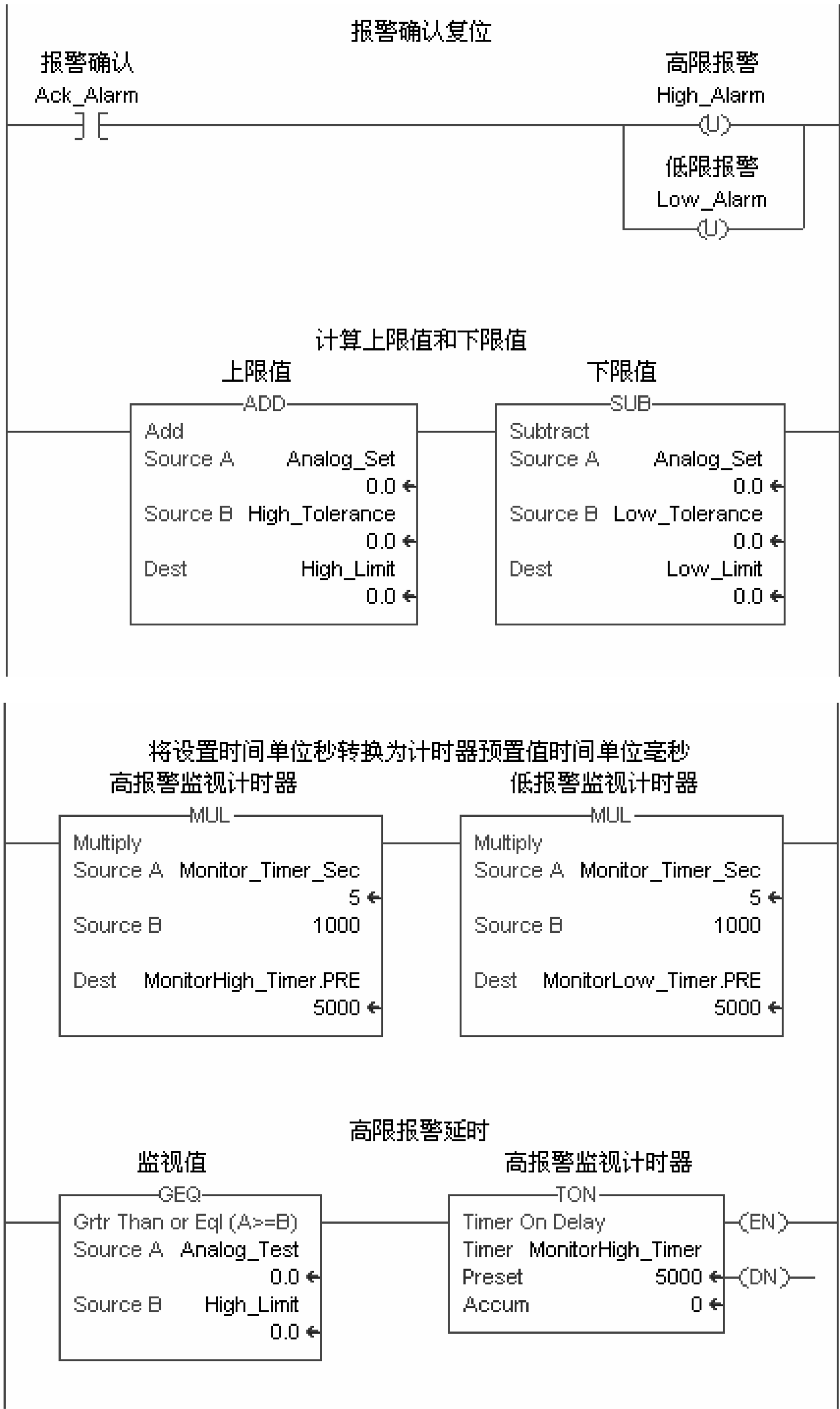


图 15-48　监视模拟量的梯级逻辑

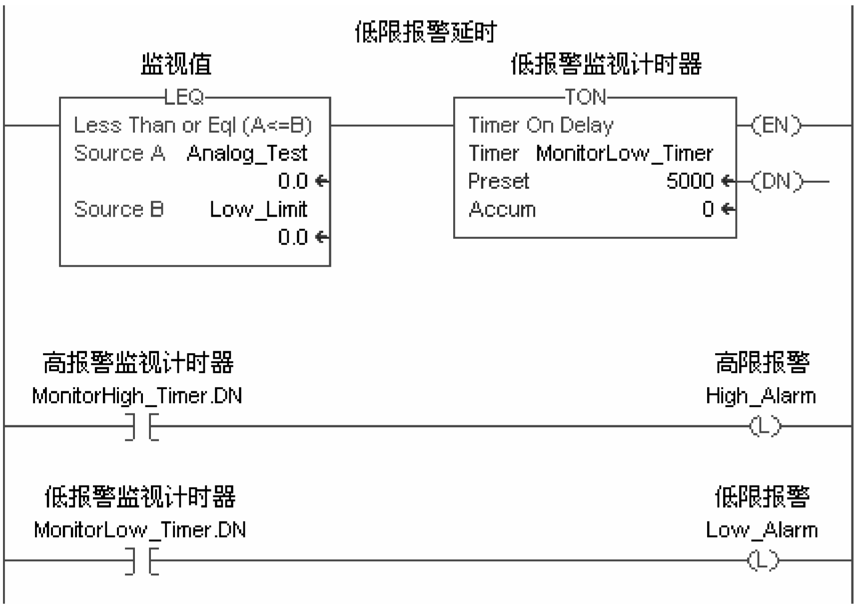


图 15-48　监视模拟量的梯级逻辑（续）

基值为毫秒。乘法指令 MUL 不但完成了时间单位的换算，同时也将设定的时间传递给了延时计时器的预置值。比较指令给定了高限计时器延时和低限计时器延时的梯级条件，当持续时间未达到延时时间而监视值退出高限值范围或低限值范围，计时器复位，下次重新进入限值范围开始计时；当持续时间达到设定的延时时间，完成位置位。计时器的完成位的置位将锁定高限报警位或低限报警位，直到报警确认复位。此外，如果希望高限报警延时时间和低限报警时间不同，可以增设低限报警时间设置值的指令参数，并分别运算不同的源数据。

编辑完成的 ANALOG 指令将出现在分类指令集的 Add_On 指令中，当我们在例程中引用该条指令时，将出现如图 15-49 所示的界面。

为本条指令分配一个结构数据标签 AnalogTest，这个结构数据标签是与 ANLOG 指令同名的结构数据，它将存放与 ANLOG 指令操作有关的全部数据，并在数据库给出了所有的指令参数，根据指令参数设置时的用法选择，有的参数是输入设定的，如 Analog_Set、High_Tolerance、Low_Tolerance 和 Monitor_Timer_Sec，有的参数用于监视并显示在 ANLOG 指令上，如 High_Alarm 和 Low_Alarm。

由于指令后台运行的结果，AnalogTest 标签的子元素是活动的数据，如果指令参数没有选择标签 AnalogTest 的子元素，而是选择了同类数据类型的其他标签，那么标签 AnalogTest 中的子元素将不断地给其他标签相互复制（拷入或拷出）。指令中测试对象直接使用了模拟量输入通道 0。从指令面板上可以看出，测试对象数据居于 15616，正好在高限值 16000 和低限值 14000 之间，故此没有任何报警状态出现。报警延时时间是默认值的 5s。

标准化的技术处理聚集了丰富的经验和严谨的结构，规范的使用不但令各种优势得以延续，还避免了新手的失误和遗漏，更可以让维护人员尽快适应例程的解读。

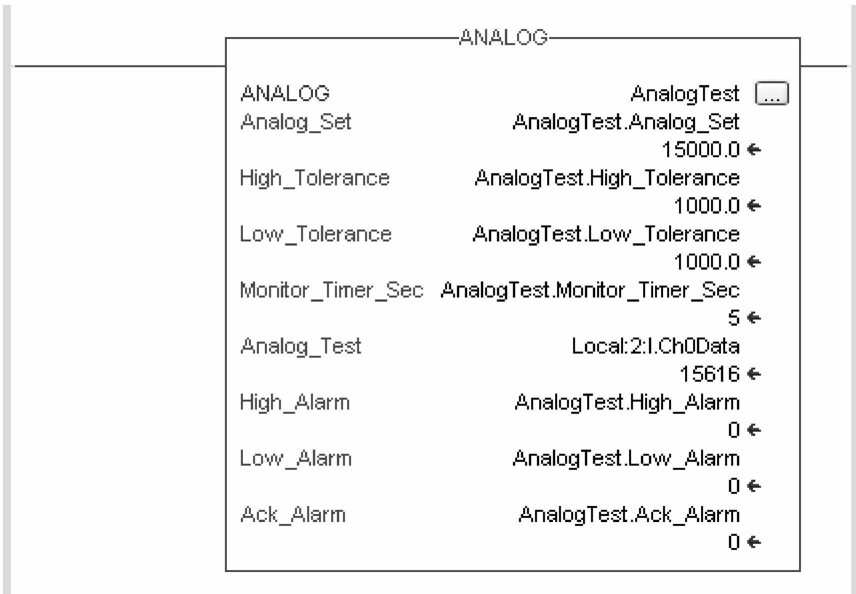


图 15-49　在例程中编写调用 ANALOG 的梯级逻辑

15.5　将特定数据处理转化为 Add-On 指令的实例

在 Logix 控制器系列中，有的模拟量模块已具备很强的数据处理能力，可以对模拟量数据进行详尽的组态，如数据定标、超限报警和限幅输出等。但是很多模块，特别是分布式的模块，数据处理的能力不是那么强，所以在很多项目的规划中都会创建一个模拟量处理的例程，专用来处理模拟量信号，编制标准的模拟量处理 Add_On 指令不失为一个好办法。下面，我们创建一个模拟量输入数据处理的 Add_On 指令和一个模拟量输出数据处理的 Add_On 指令。

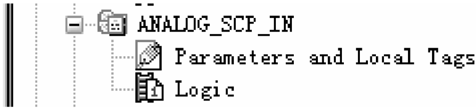


图 15-50　创建一个模拟量输入定标指令

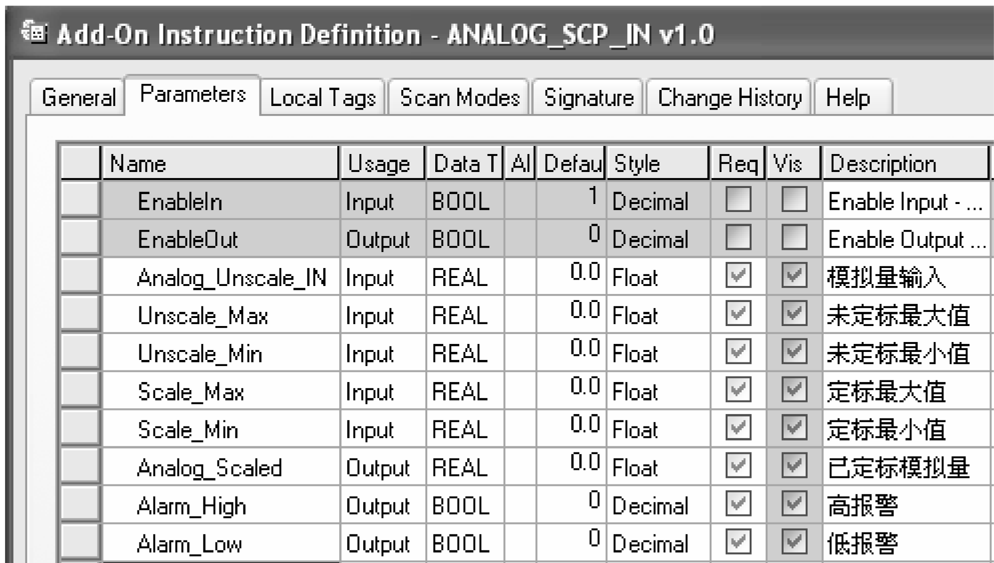
创建一个模拟量输入定标指令 ANALOG_SCP_IN，如图 15-50 所示。

创建如图 15-51 所示的指令参数设置，各个指令参数的用法和数据类型以及说明如表中所示，并设定参数显现在指令中。

创建如图 15-52 所示的本地标签，用来存放偏移量和比例系数中间结果的数据单元。

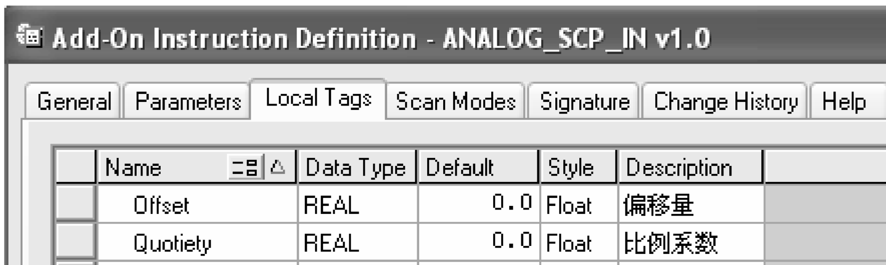
在 Logic 例程中编写梯级逻辑如图 15-53 所示，使用 CPT 指令计算比例系数和偏移量以及定标输出的运算。CPT 指令是最常见的综合运算指令，按照算术逻辑运算规则书写的表达式可以将所需的运算综合在一起，使用一条指令就让人一目了然计算公式和运算目的。这条指令虽然表达明了，但是较为消耗时间，所以单一的运算并不推荐使用这条指令，而是建议直接使用单一的运算指令，如前面所使用的加法指令、减法指令和乘法指令。

使用大于等于指令和小于等于指令来判定是否进入最高定标或最低定标的范围，定标报警信号使用的非保持型继电器输出指令，这里只是表达了当时的一种状态，并不要锁定报警状态，所以无须延时或锁定。如果需要锁定报警状态，可以将这个进入定标范围外的状态另



Name	Usage	Data T	Al	Defau	Style	Req	Vis	Description
EnableIn	Input	BOOL		1	Decimal	☐	☐	Enable Input - ...
EnableOut	Output	BOOL		0	Decimal	☐	☐	Enable Output ...
Analog_Unscale_IN	Input	REAL		0.0	Float	☑	☑	模拟量输入
Unscale_Max	Input	REAL		0.0	Float	☑	☑	未定标最大值
Unscale_Min	Input	REAL		0.0	Float	☑	☑	未定标最小值
Scale_Max	Input	REAL		0.0	Float	☑	☑	定标最大值
Scale_Min	Input	REAL		0.0	Float	☑	☑	定标最小值
Analog_Scaled	Output	REAL		0.0	Float	☑	☑	已定标模拟量
Alarm_High	Output	BOOL		0	Decimal	☑	☑	高报警
Alarm_Low	Output	BOOL		0	Decimal	☑	☑	低报警

图 15-51　创建指令参数



Name	Data Type	Default	Style	Description
Offset	REAL	0.0	Float	偏移量
Quotiety	REAL	0.0	Float	比例系数

图 15-52　创建本地标签

外编程来解决。

注意到，我们只是运算出定标的换算关系，并没有限制信号的范围，所以运算出来的结果很有可能是超出定标范围的。本条 Add_On 指令只如实完成定标的工作，输入是不会限幅的，但是要报告超出定标范围的状态，让外部的程序决定该怎样处理。

编辑完成的 ANALO_SCP_IN 指令将出现在分类指令集的 Add_On 指令中，当我们在例程中引用该条指令，将出现如图 15-54 所示的界面。

为本条指令分配一个结构数据标签 Analog_SCP_In1，这个数据标签是与 ANALO_SCP_IN 指令同名的结构数据类型，它将存放与指令操作有关的全部数据，并在数据库给出了所有的指令参数。根据指令参数设置时的用法选择，有的参数是输入设定的，如 Unscale_Max、Unscale_Min、Scale_Max 及 Scale_Min，有的参数用于监视并显示在指令上，如 Analog_Scaled、Alarm_High 和 Alarm _Low。

由于指令后台运行的结果，Analog_SCP_In1 标签的子元素是活动的数据，如果指令参数没有选择标签 Analog_SCP_In1 的子元素，而是选择了同类数据类型的其他标签，那么标签 Analog_SCP_In1 中的子元素将不断地给其他标签相互复制（拷入或拷出）。指令中针对模拟量 1 通道定标，将模拟量信号的 A/D 转换结果 0 ~ 31140 范围转换为定标值 1 ~ 10000 的

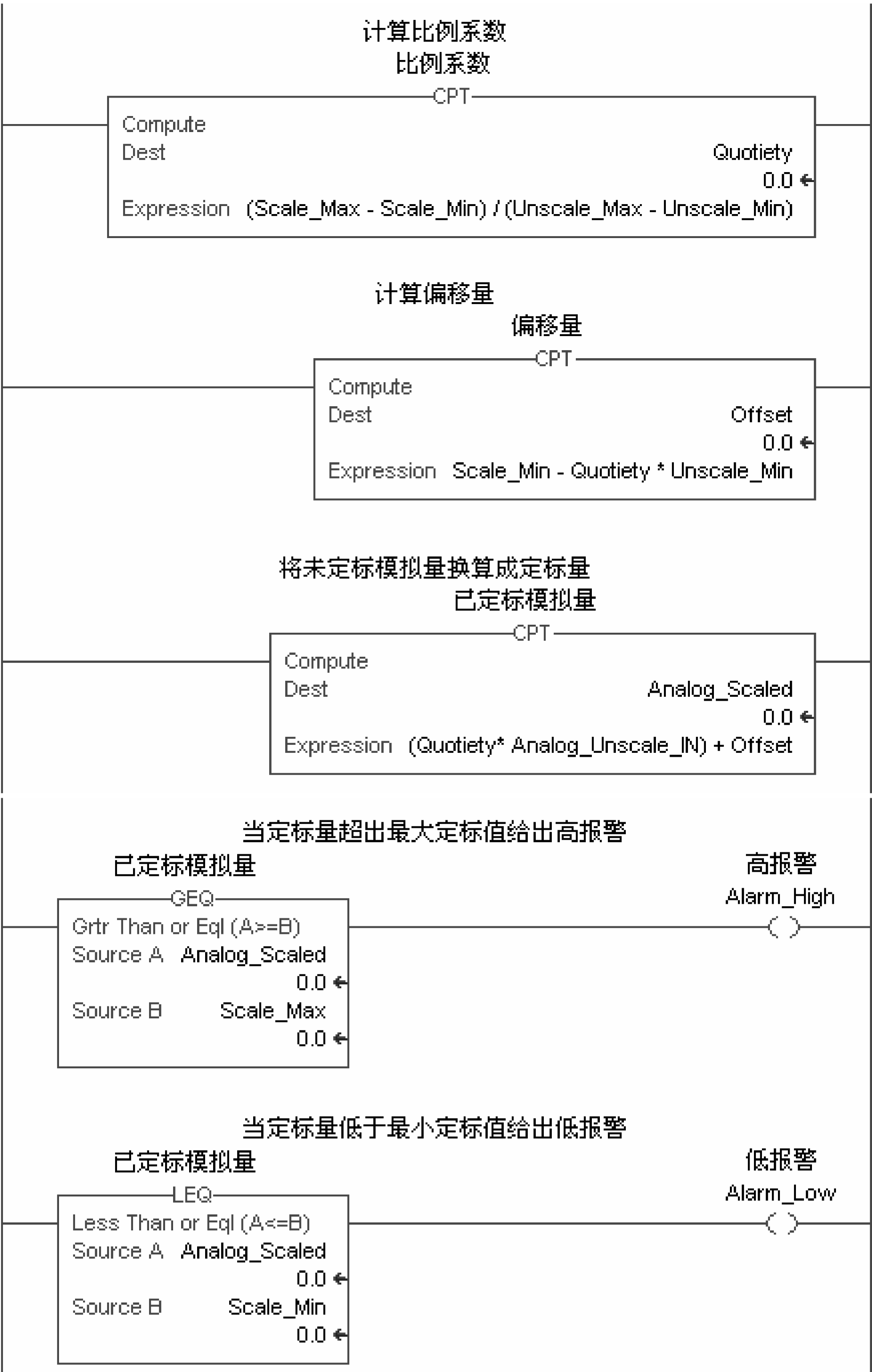


图 15-53 定标计算的梯级逻辑

范围。可以看到输入量 4224 定标为 1356. 4547 的值，定标后的值将被执行代码引用。

创建一个模拟量输出定标指令 ANALOG_SCP_OUT，如图 15-55 所示。

创建如图 15-56 所示的指令参数设置，各个指令参数的用法和数据类型以及说明如表中

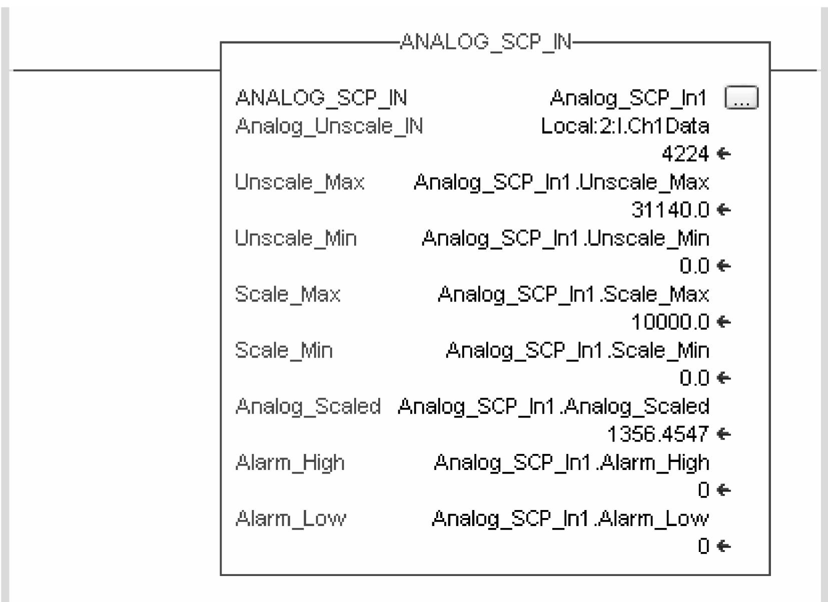


图 15-54　在例程中编写调用 ANALOG_SCP_IN 指令的梯级逻辑

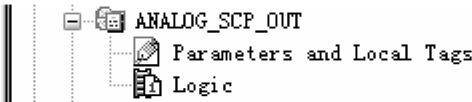


图 15-55　创建一个模拟量输出定标指令

所示，并设定参数显现在指令中。

Add-On Instruction Definition - ANALOG_SCP_OUT v1.0

General | Parameters | Local Tags | Scan Modes | Signature | Change History | Help

	Name	Usage	Data T	Al	Defau	Style	Req	Vis	Description
	EnableIn	Input	BOOL		1	Decimal	☐	☐	Enable Input - System ...
	EnableOut	Output	BOOL		0	Decimal	☐	☐	Enable Output - Syste...
	Analog_Unscale	Input	REAL		0.0	Float	☑	☑	未定标值
	Unscale_Max	Input	REAL		0.0	Float	☑	☑	未定标最大值
	Unscale_Min	Input	REAL		0.0	Float	☑	☑	未定标最小值
	Scale_Max	Input	REAL		0.0	Float	☑	☑	定标最大值
	Scale_Min	Input	REAL		0.0	Float	☑	☑	定标最小值
	Limit_High	Input	REAL		0.0	Float	☑	☑	高限幅值
	Limit_Low	Input	REAL		0.0	Float	☑	☑	低限幅值
	Analog_Scaled_OUT	Output	REAL		0.0	Float	☑	☑	已定标模拟量输出值
	Alarm_High	Output	BOOL		0	Decimal	☑	☑	高报警
	Alarm_Low	Output	BOOL		0	Decimal	☑	☑	低报警

图 15-56　创建指令参数

创建如图 15-57 所示的本地标签，用来存放计算结果、偏移量和比例系数中间结果的数据单元。

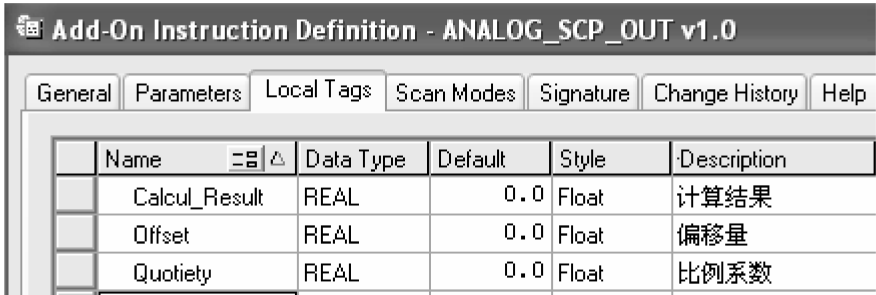


图 15-57 创建本地标签

编写梯级逻辑如图 15-58 所示，同样是用 CPT 指令来完成偏移量和比例系数的计算，但是定标计算后的结果不能直接送出，模拟量输出的定标指令应该具有限幅的功能，以防止信号输出超幅，从而保护外部回路。也许有的模拟量输出模块本身就具备限幅功能，作为通用于所有的模拟量输出的定标指令，我们必须编写逻辑关系来进行限制。

在判断进入限幅范围时，令输出限制在最大限幅或最小限幅上，同时给出限幅警告。未进入限幅范围，则按定标结果给予输出。

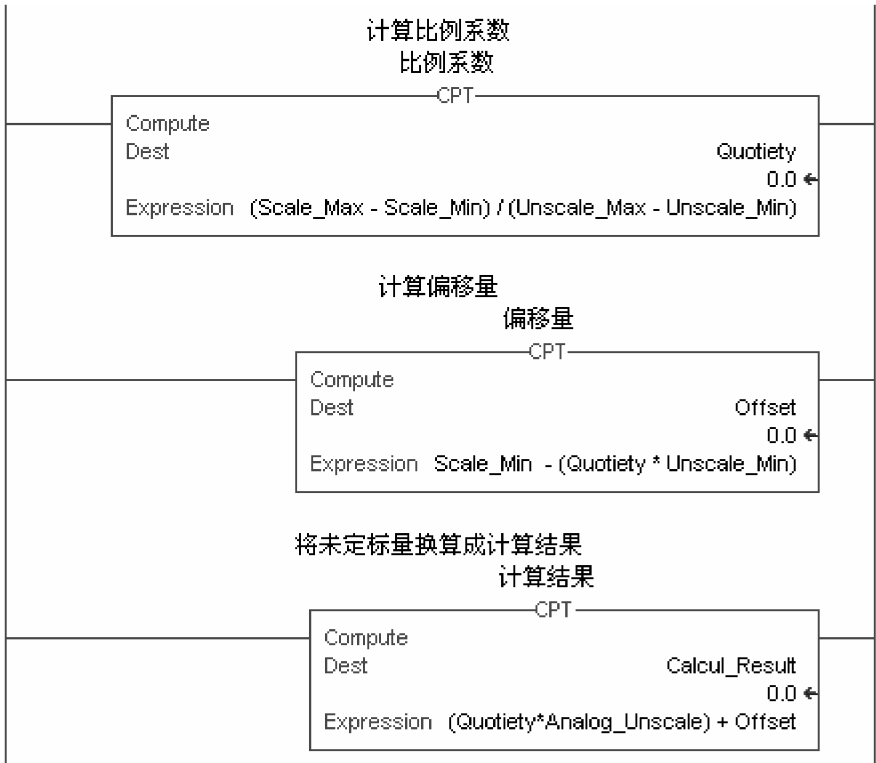


图 15-58 定标计算的梯级逻辑

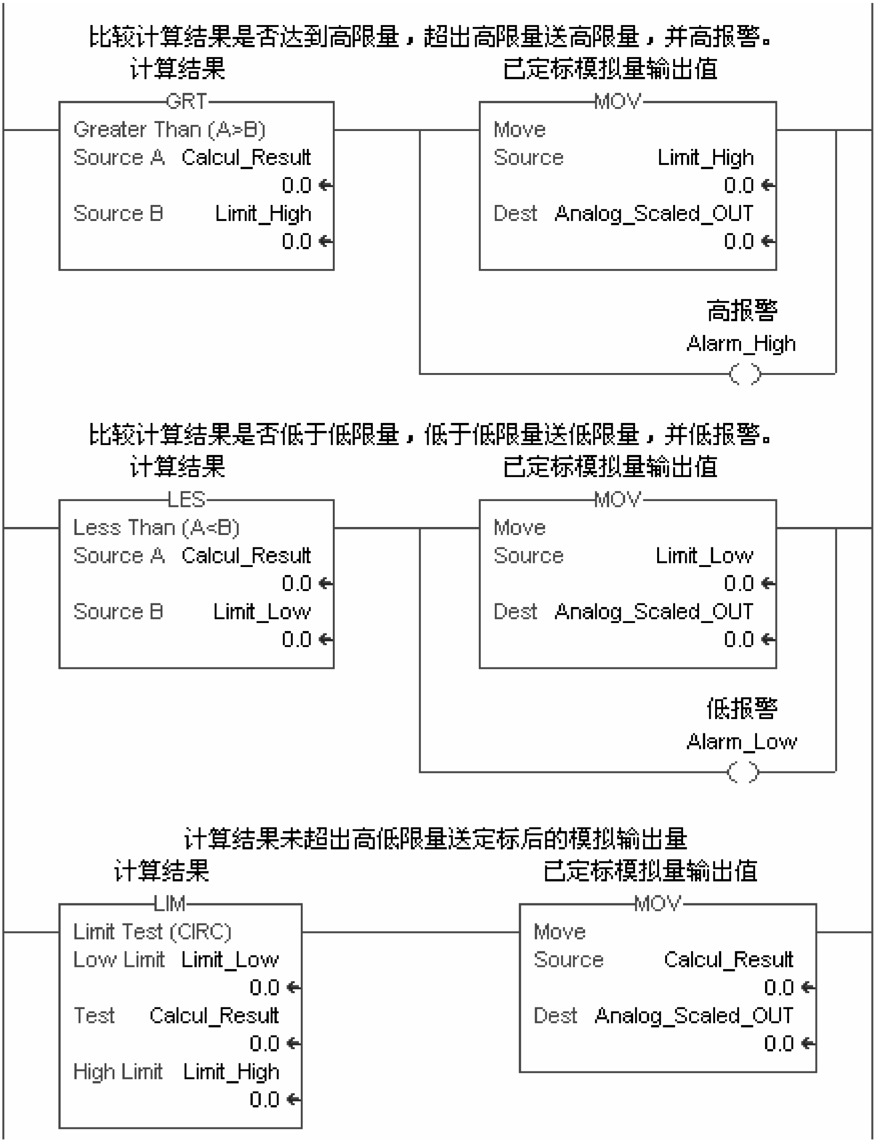


图 15-58　定标计算的梯级逻辑（续）

大于比较指令 GRT、小于比较指令 LES 和范围测试指令 LIM 分辨了输出的范围，这里 LIM 指令 High Limit 和 Low Limit 的比较范围是包含上限值和下限值本身的数值的，所以只能采用大于而不是大于等于，采用小于而不是小于等于，以免分界值模糊不清，并将 LIM 指令的梯级放置于 GRT 指令和 LES 指令的梯级之后执行。

编辑完成的 ANALO_SCP_OUT 指令将出现在分类指令集的 Add_On 指令中，当我们在例程中引用该条指令，将出现如图 15-59 所示的界面。

为本条指令分配一个结构数据标签 Analog_SCP_Out1，这个数据标签是与 ANALO_SCP_OUT 指令同名的结构数据，它将存放与指令操作有关的全部数据，并在数据库给出了

所有的指令参数。根据指令参数设置时的用法选择，有的参数是输入设定的，如 Unscale_Max、Unscale_Min、Scale_Max、Scale_Min、Limit_High 及 Limit_Low，有的参数用于监视并显示在指令上，如 Analog_Scaled_OUT、Alarm_High 和 Alarm _Low。

由于指令后台运行的结果，Analog_SCP_Out1 标签的子元素是活动的数据，如果指令参数没有选择标签 Analog_SCP_Out1 的子元素，而是选择了同类数据类型的其他标签，那么标签 Analog_SCP_Out1 中的子元素将不断地给其他标签相互复制（拷入或拷出）。指令中将程序运行的结果送至子元素 Analog_SCP_Out1. Analog_Unscale 中，这是一个 0 ~ 10000 范围的数值，经过定标转换为模拟量输出模块 D/A 转换所需要的数据范围 0 ~ 31140。从指令中可以看到，从 3300 转换为 10276 的输出数据将送至模拟量输出通道 1，然后 D/A 转换为相应的模拟量信号。此处数据在限幅范围之中，没有报警，输出也未限幅。

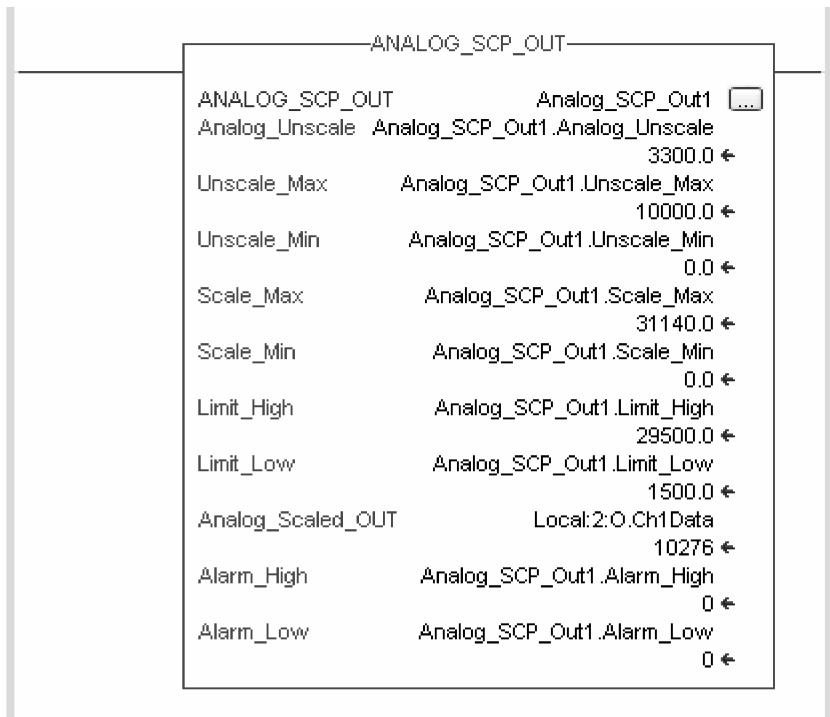


图 15-59　在例程中调用 ANALOG_SCP_OUT 指令的梯级逻辑

大家也许注意到了前面三条关于数据标准化处理的 Add_On 指令，在指令界面参数的排列不是随意的，不但归类、对称、具有层次感，还要排列为输入数据在前输出数据在后，因果关系明了。如果感觉原先的排列不合适，参数位置的排列可在 Add_On 指令的参数页面用 Move Up 或 Move Down 调整，如图 15-60 所示。

本章中，我们列举了多个熟悉的实例，转化成为 Add_On 指令的执行，可以说 Add_On 指令实际上就是将实例转化为用户自定义的常规指令动作，嵌入指令系统中共享。因而，在 RSLogix5000 的编程软件中，Add_On 指令的导入和导出都十分方便，并以独立文件形式存在，便于发布和交换。

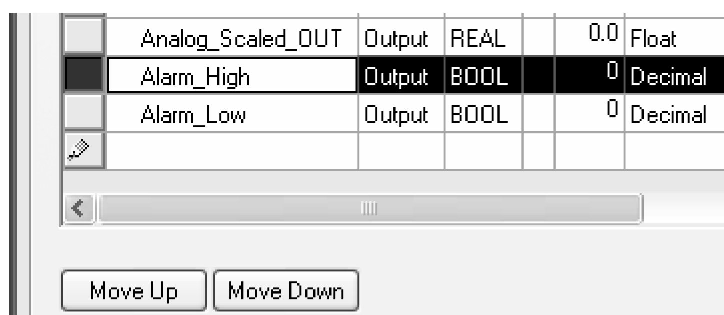


图 15-60　可调整指令参数位置

第 16 章

PhaseManager 编程介绍

PLC 编程发展的较高阶段就是标准化的编程，无论从缩短工程开发周期，节约成本，还是后期的系统维护方便，都具有非常重大的实际意义。

Logix 控制器为标准化编程提供了优良的平台，描述性的数据标签、用户自定义结构数据、多层的程序结构、灵活的编程方式、用户自定义指令、直观的监视界面等。

任何控制器系统的时序逻辑控制最终都是设备动作的执行进程，S88 就是一种针对批量配方处理过程的设备执行标准的模式，它将设备工作进程标准化，并建立起固定的结构。PhaseManager 是将这种设备进程工作标准用程序实现的编程模板，状态模块将设备工作进程结构化，并在每个指定的状态框中编写相应的执行逻辑。

控制器内置了一个用于管理设备阶段进程的状态模块，所做的事情是：

- 为每一个激活状态调用状态例程；
- 管理状态之间的转换，当梯级条件成立，转入必需的下一状态；
- 确定设备沿着固定的路径从一个状态转换到另一个状态，锁定了状态之间的关系。

以此获得了标准化的操作过程。

16.1　PhaseManager 的创建

PhaseManager 跟程序一样，也是属于任务下的结构，选择任务，点击右键，如图 16-1 所示，选择与程序平行的 Equipment Phase，其结构和作用跟程序是相似的。

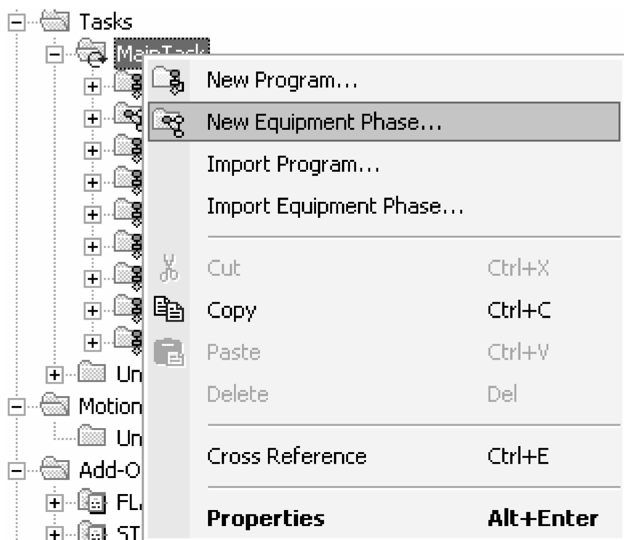


图 16-1　选择一个新的 Equipment Phase

点击“Equipment Phase”，进入创建页面，为其命名为 Phase，如图 16-2 所示。在这个页面还可以再次安排在另外的任务之下。

选中已创建的 Phase，点击右键，创建新的状态例程，如图 16-3 所示。

点击“New Phase State Routine”，进入创建页面，如图 16-4

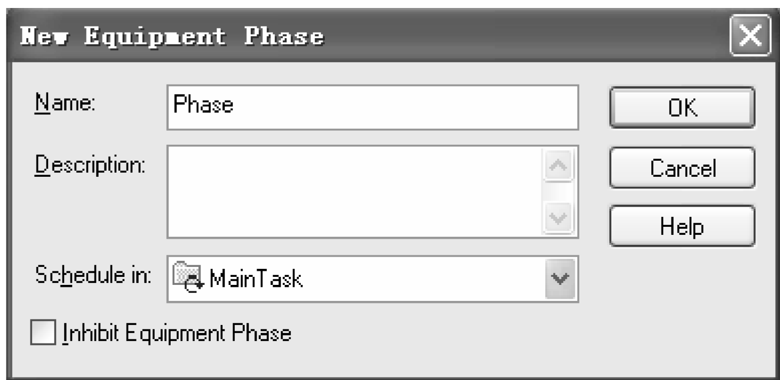


图 16-2　命名为 Phase

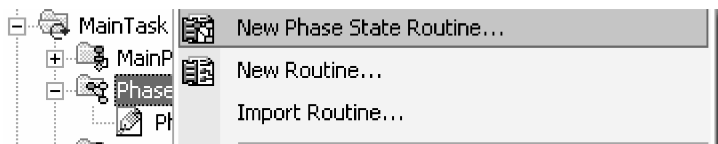


图 16-3　创建新的状态例程

所示的是可选择的状态例程，一共 6 个，每个状态例程只有 1 个，并且只能被创建 1 次。

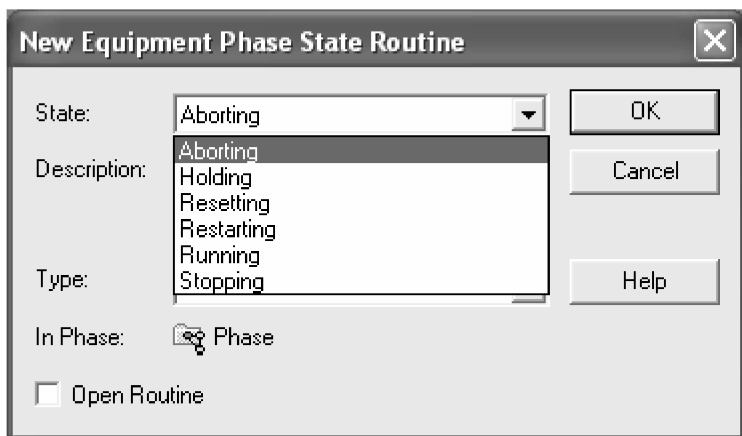


图 16-4　可选择的状态例程

选择状态例程，如此重复 6 次，直到创建了所有的状态例程，如图 16-5 所示。这些状态例程将作为状态模块的主要进程，并在例程中编写相关操作的梯级逻辑。

选中 Phase，点击右键，如图 16-6 所示。

点击 "Monitor Equipment Phase"，进入状态模块监视界面，如图 16-7 所示。

设备的状态过程在这个监视画面里一览无余，可以看到刚才创建的 6 个状态例程，一个状态例程过渡到下一个状态例程时，总是要经过一个预状态的过程，再从预状态进入下一个状态例程。例如 Running 状态例程后，进入 Complete 预状态，只

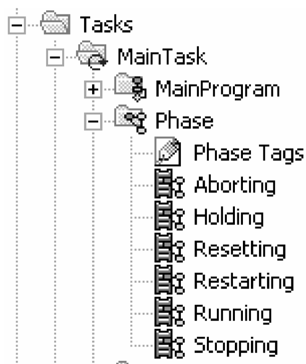


图 16-5　已创建的状态例程

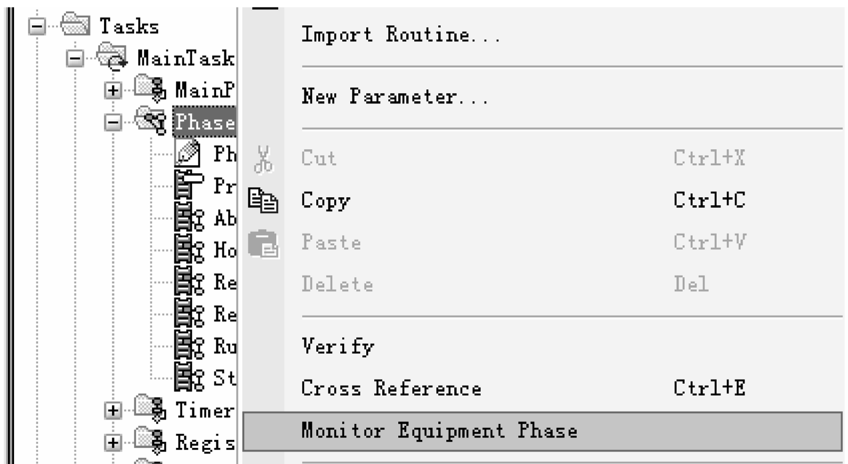


图 16-6　展开 Phase

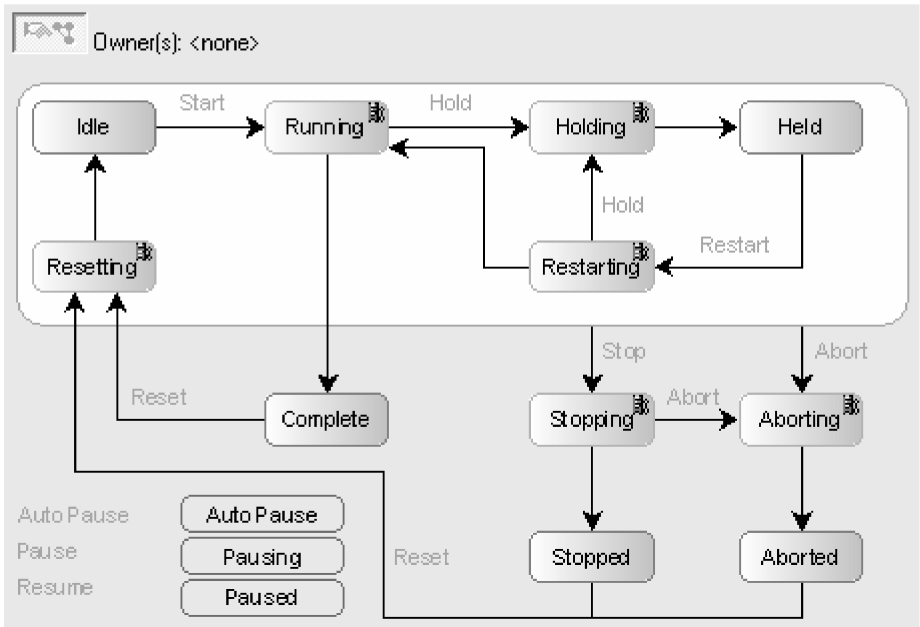


图 16-7　状态模块监视界面

有命令 Reset 的执行才能进入下一个状态例程 Resetting，离开 Resetting 状态例程后，进入预状态 Idle，执行命令 Start，则进入 Running 状态例程，这刚好是一个正常的工作循环。

Reset 命令和 Start 命令的发送是一条特别的指令执行结果，这条指令执行必须放置在时时被扫描的一个例程中，这就是预状态例程，它的作用类似于程序中的主控例程，不可或缺。如图 16-8 所示，预状态例程总是和状态模块中的某个状态例程交替进行，保持着连续的扫描，以确保梯形逻辑能够得到及时的执行。执行一些重要命令的指令都要编写在预状态例程中。

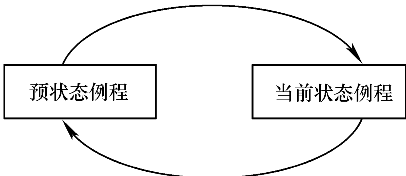


图 16-8　预状态例程总是和某个状态例程交替进行

下面来创建一个预状态例程。

选中 Phase，点击右键，如图 16-9 所示。

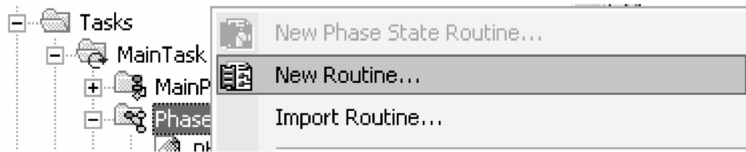


图 16-9　选中 Phase 点击右键

点击“New Routine”，创建一个普通的例程，命名为 PreState，如图 16-10 所示。

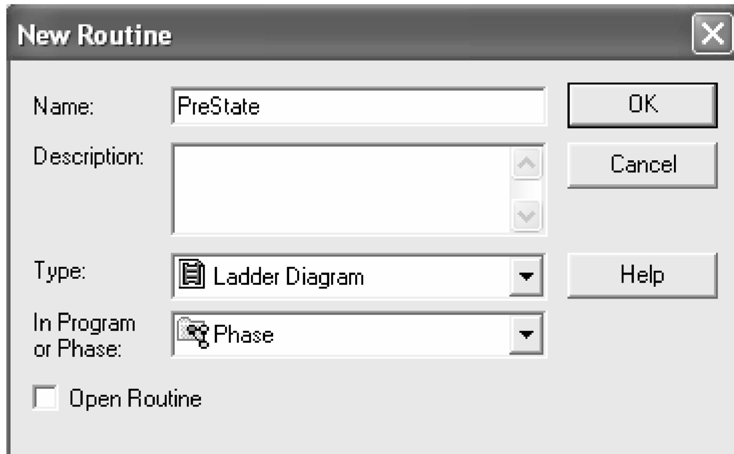


图 16-10　创建一个普通的例程

从 Phase 的属性进入组态页面，在 Prestate 项中选择刚才创建的例程作为预状态例程，如图 16-11 所示。

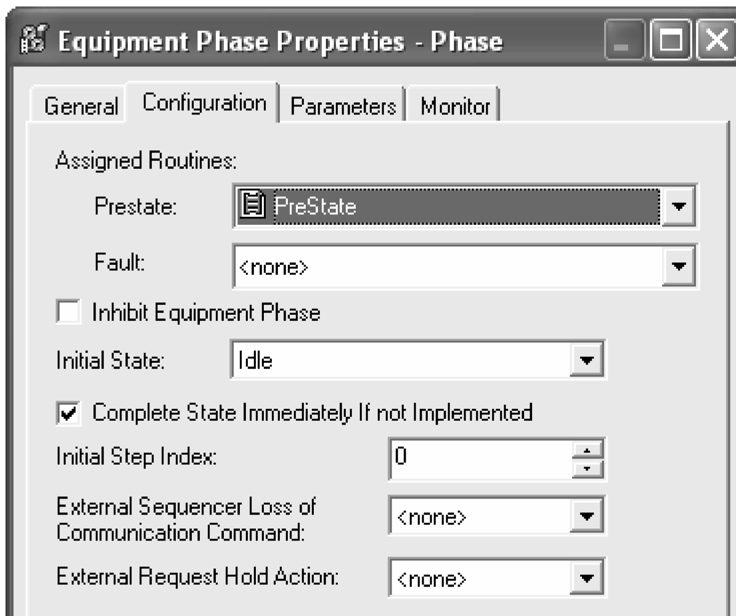


图 16-11　在 Prestate 项中选择预状态例程

经过组态的选定后，产生了预状态例程，如图 16-12 所示。

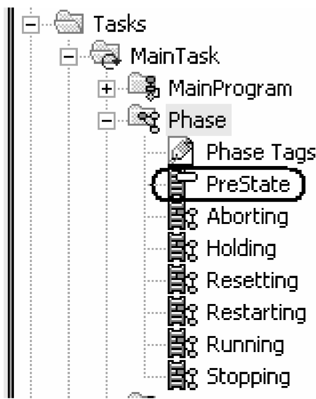


图 16-12　添加了预状态例程

至此，才是一个较为完整的创建。

16.2　PhaseManager 的编程

下面我们用一个惯用的简单例子来经历编程的整个过程。用 4 个轮流闪烁的灯来模拟某设备的 4 个工作过程，令每个灯闪烁 20s 后完成一个工作过程，转入下一个状态进程。

在 Running 状态例程是专门用来编写生产过程的，按照灯闪烁的模拟过程，编写如图 16-13 所示的梯形逻辑，用自复位计时器 Timer_Phase_RUN 产生 20s 的周期时间，计数器 Counter_Phase_RUN 计数且作为指针，运用先前创建的 Add_On 指令 FLASH 完成灯的闪烁动作。当计数器计数 4 次，完成位置位，模拟了生产过程一个周期的完成。

计数器的完成位作为梯级条件，执行 PSC 指令，PSC 指令是专用来退出状态例程的，要求编写在每个状态例程的最后一个梯级，以保证完整地执行整个状态例程。PSC 指令的执行不但退出当前状态例程，并令退出的状态例程静止，即复位所有的非保持型指令。这个例程的静止并不能将计数器指针复位，如不进行相应的处理，一旦执行 Start 命令进入新的工作周期，计数器的完成位会立即执行 PSC 指令，刚进入的 Running 状态例程将迅速退出。归属于下一个工作周期的启动准备工作，理应在 Resetting 状态例程中解决计数器复位这个问题，所以在这个例程中暂不做处理。

从状态模块可以看出，Running 状态例程的启动，需要从 Idle 的预状态执行 Start 命令才能进入。在预状态例程中编写如图 16-14 所示的梯形逻辑。PCMD 是设备阶段管理的命令指令，借助于这条指令的执行，可以发送 Start、Hold、Restart、Stop、Abort 和 Reset 等命令，这些我们在后面都会用到。

PCMD 指令参数 Result 是一个双整字，用来记录本条指令执行的结果，其编码代号说明了某种执行状态，结果为 0 表示指令成功执行。如果你看到的 Result 数据标签中显示的不是 0 而是 24577，代码 24577 表示命令执行无效，这并不表明命令没有执行成功，也许你编制的命令指令 PCMD 没有严格地确保只执行一次，在进入 Running 状态例程之后，由于梯级条件依然存在，即使是你认为的短暂的时间（如按下后释放），命令指令又被执行了多次，由

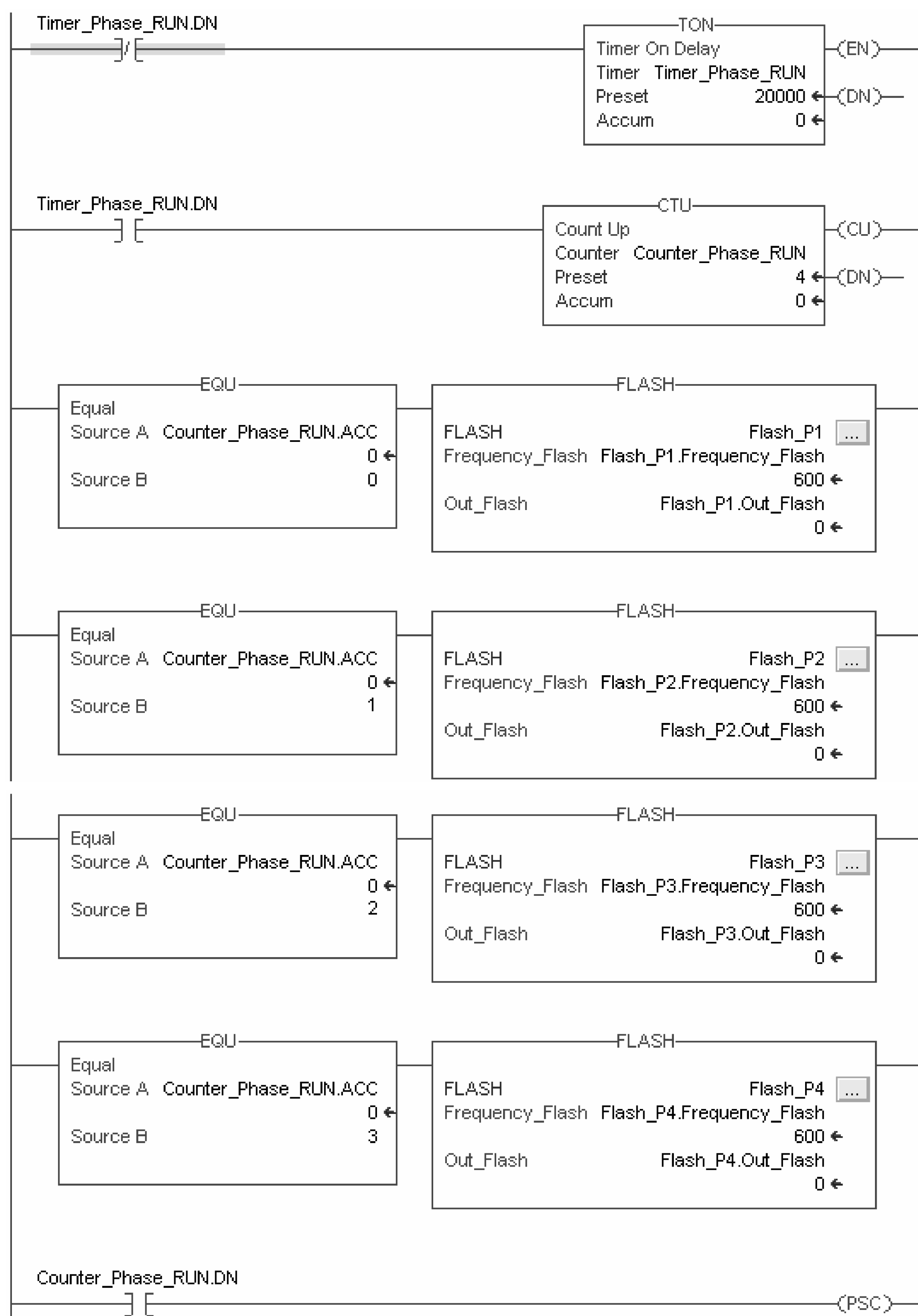


图 16-13　Running 例程的梯形逻辑

于已经不在合理的状态例程进程中，命令执行自然是无效的，只有在 Idle 预状态时执行 Start 的命令才是合理的状态进程。如果你希望用这个执行代码的结果来证实命令确被执行，并用作别处的依据，请考虑以上原因，或采用 ONS 指令予以限制。

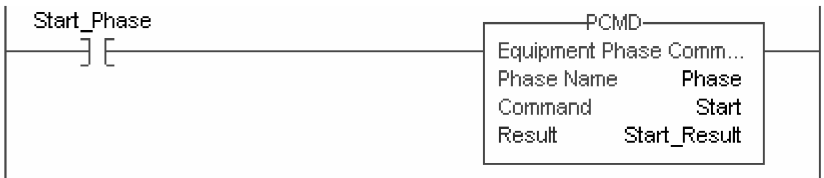


图 16-14　执行启动命令的梯形逻辑

编辑在预状态例程中的这个梯级，被连续扫描，一旦梯级条件成立，PCMD 指令执行，Start 命令发出，进入 Running 状态例程，可以测试一下生产过程。Running 状态例程中，应该编制所有的生产工艺过程的梯形逻辑，本例中是 4 个灯轮流闪烁一遍。当 Running 状态例程完成整个工艺过程，PCS 指令执行，将进入预状态 Complete 等待下一个状态例程。从状态模块上的进程可以看出，下一个状态例程是 Resetting 状态例程。

至此，尚未完成整个的工作周期，在预状态例程中，还要编写一条从预状态 Complete 进入状态例程 Resetting 的命令，如图 16-15 所示。

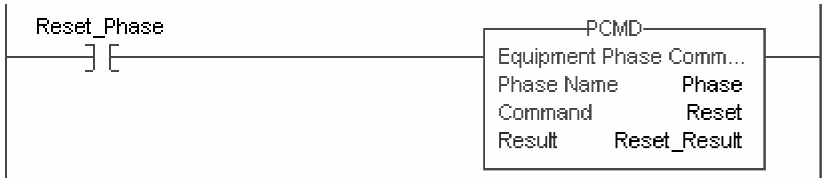


图 16-15　执行复位命令的梯级逻辑

当梯级条件成立，命令指令执行 Reset 的结果，进入 Resetting 状态例程中，此为进入正常工作进程的必由之路。在 Resetting 状态例程中，主要编写生产工艺过程进入的准备工作的梯级逻辑，我们模拟一个准备工作过程，编写梯形逻辑如图 16-16 所示。

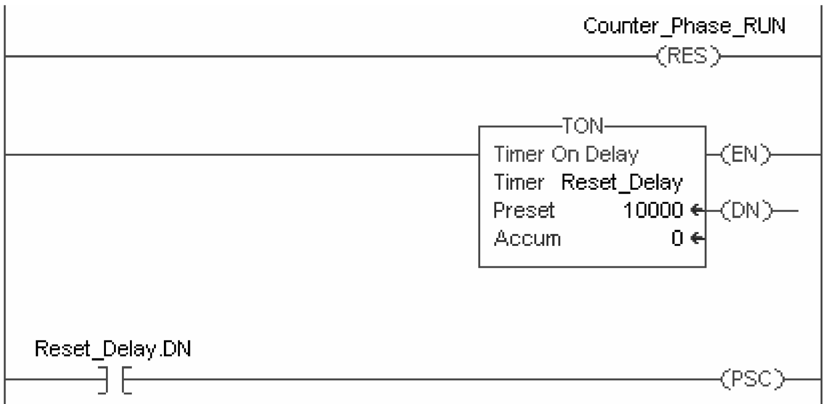


图 16-16　准备工作的梯形逻辑

首先，给作为生产工艺进程指针的计数器 Counter_Phase_RUN 复位，以便进入运行例程后将从头开始工作。然后用一个延时 10s 的计时器模拟准备工作所需要花费的时间，生产

过程的准备不仅仅是电器设备状态的清除或预设，有的机械设备也可能需要回位，所以一般都需要一些延时时间。等延时 10s 到，计时器 Reset_Delay 完成位提供的梯级条件引起 PSC 指令的执行，退出当前状态例程，并令所有梯级静止，即所有非保持型指令复位，这会将计时器复位，不必担心下次进入 Resetting 状态例程时计时器是否能重新开始计时。

现在我们注意到了，从 Running 状态例程退出和 Resetting 状态例程退出可知，分别对计时器指令和计数器指令的作用是不相同的。这里再次强调，PSC 指令的作用有两个，一是退出当前状态例程，二是令状态例程静止，即复位所有的非保持型指令。在各个状态例程中编写执行代码时，一定要考虑 PSC 指令执行对状态例程中某些数据的影响。这跟我们之前一直强调预扫描和后扫描对例程数据的影响道理是一样的，但是因为状态例程是一个重复进行的过程，数据是否被复位就显得尤为重要。

此外，还要提请注意的是，PSC 指令一定要编写在状态例程的最后一个梯级，以避免不完整的例程扫描，因为一旦 PSC 指令执行，即离开例程，不再继续扫描。

现在，我们可以让整个生产工艺周期完整地运行起来了，这个进程在状态模块中观察是最方便和直观的。

为什么启动运行从 Start 命令开始呢？从状态模块中我们可以看到，系统进入运行的初始状态是停留在 Idle 的预状态位置，这是常用的模式，也是默认的组态，如图 16-17 所示。可以看到，根据不同的需求，也可以设置成另外的预状态。有关其他组态的定义，这里不作深入讨论，请参考《ControlLogix 系统实用手册》一书。

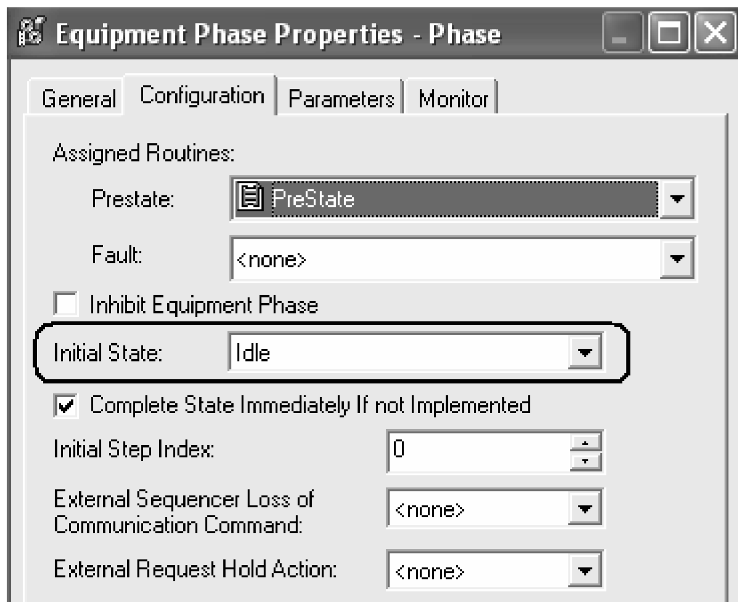


图 16-17　默认的预状态组态

刚才我们讨论的是控制系统正常运行循环的程序编写，异常情况的处理有专门的状态例程来处理，这恰恰是标准化编程的特点，正因为如此，才体现出标准化编程给后期的系统维护带来的方便和功能优势。

当设备临时性地暂停生产过程，进行某些处理之后，重新回到刚才中断的进程继续工作，我们需要在预状态例程中编写一条命令指令，用来脱离当前的 Running 状态例程，进入

暂停状态例程 Holding，如图 16-18 所示。

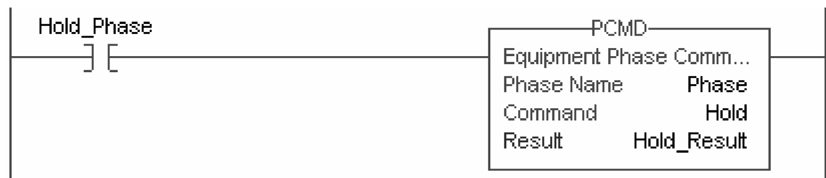


图 16-18　执行暂停命令的梯级逻辑

进入暂停状态例程，相关的处理完毕之后，要回到中断的工作点。从状态模块上我们看到，要通过暂停结束预状态例程 Held 和重启状态例程 Restarting 才能回到 Running 状态例程。

在暂停状态例程 Holding 中，编写梯形逻辑如图 16-19 所示。无条件梯级点亮一个 Holding 状态的灯，显示目前的暂停状态，当状态例程离开时，由于 PSC 指令的静止作用，也即状态例程的后扫描，非保持型的输出将复位为 0，状态灯 Holding_Light 自然熄灭。给出退出 Holding 的条件 Go_Held 后，Holding 状态例程退出，并进入预状态 Held。

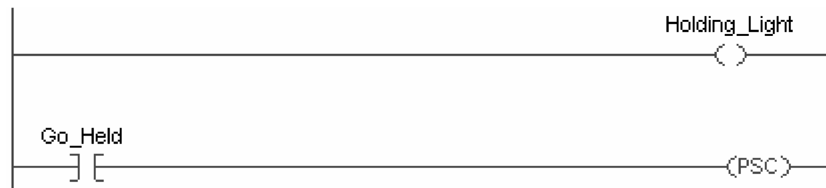


图 16-19　处在暂停状态的梯形逻辑

从状态模块中看到，只有命令 Restart 的执行才能进入回归的必由之路重启状态例程 Restarting。同样地，在预状态例程中编写命令指令，如图 16-20 所示。

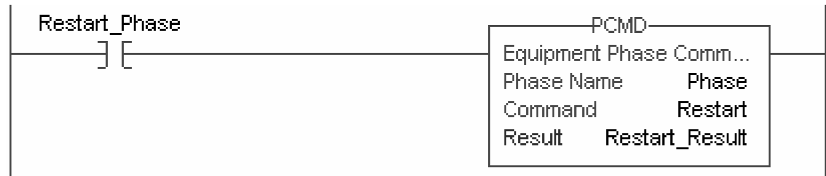


图 16-20　执行重启命令的梯级逻辑

Restart 命令的执行，可以看到状态模块中的 Restarting 的状态例程处在了激活状态，显然必须在这个状态例程中编写退出本身状态例程的 PSC 指令，直接回到 Running 状态例程中。在 Restarting 状态例程中编写梯级逻辑，如图 16-21 所示。

无条件梯级控制的重启状态显示灯 Restart_Light 在离开重启状态例程 Restarting 时，由于例程静止作用会自然熄灭。在状态模块中可以看到还有回到暂停状态例程 Holding 的机会，如果还需要进行其他处理的话，可以在这个例程中执行 PCMD 指令来发送 Hold 命令，重新回到状态例程 Holding，看来 PCMD 指令并不是只能在预状态例程中才能编写。用 Go_Running 梯级条件离开重启状态例程 Restarting，并回到 Running 状态例程。

刚才，我们又完成了一个暂停循环的编写。

将刚才编写好的程序运行起来测试一下，建议在我们模拟的 4 个灯闪烁进程的中途，例如第二个进程的灯在闪烁时进入暂停的过程，经过 Restarting 重新回到 Running 状态例程，

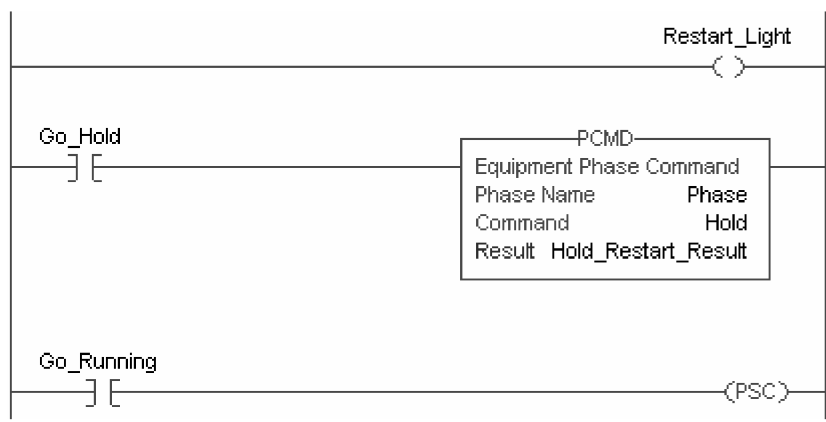


图 16-21　重启的梯级逻辑

你会发现又回到刚才中断的地方继续运行，这种暂时的离开没有后扫描的作用，也不会复位非保持型的指令。

暂停的状态是一个不会破坏生产工艺处理过程的行为。

现在，让我们重新审视状态模块，如图 16-22 所示，可以发现，刚才我们编写的正常工作的循环和暂停的循环，基本落在白色的框中，这是不遭受破坏性停机的过程。

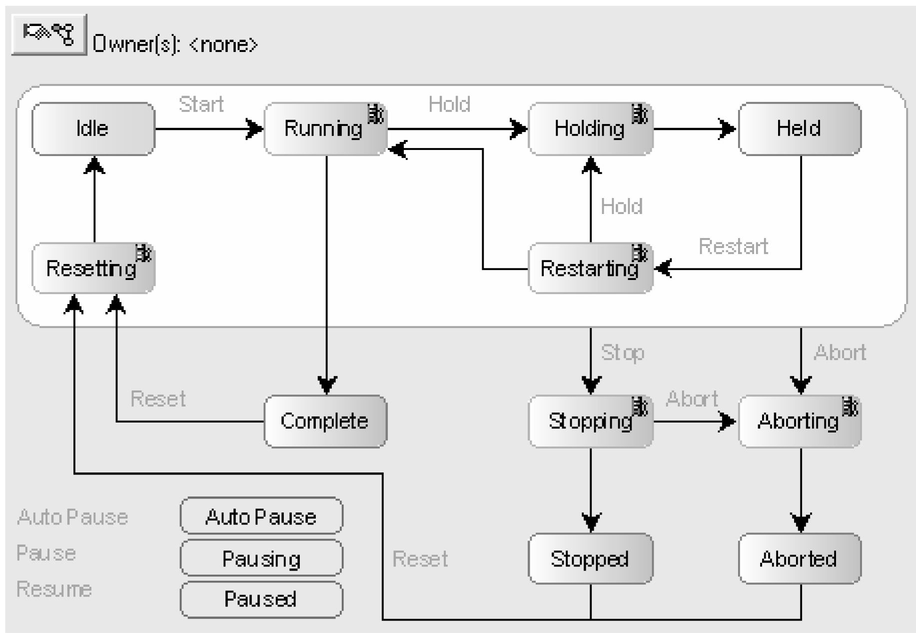


图 16-22　状态模块

注意到白色的框外接的两条命令，Stop 和 Abort，这是不受进程的限制，在任何一个状态例程或预状态都可以干预的命令。这两条命令的执行，生产工艺工程将脱离正常的进程而进入 Stopping 状态例程或 Aborting 状态例程。显然，这样的处理是较为严重的，且不能重新回到 Running 状态例程，接续曾经的中断。

在预状态或任何状态例程都可以编写这两条命令，如图 16-23 所示，这两条命令的执行将分别进入停止状态例程 Stopping 或放弃状态例程 Aborting。

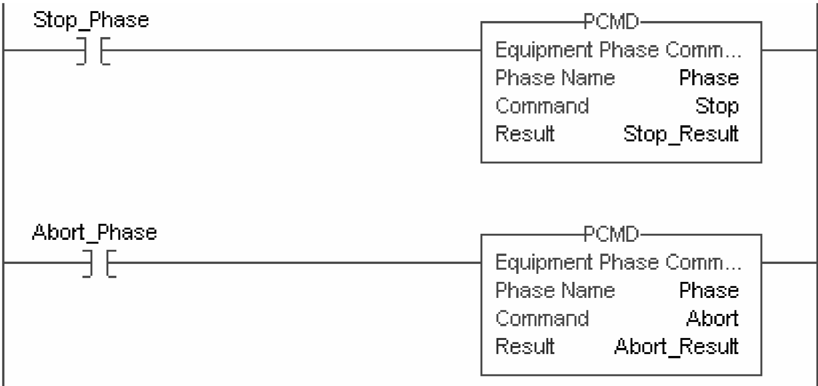


图 16-23　可以在预状态或任何状态例程编写这两条命令

停止状态例程 Stopping 处理的是正常关机、上电时的故障或者控制系统关机，通过这个状态例程来实现。从状态模块上我们可以看到，除了直接进入关机完毕的预状态 Stopped，还可以通过命令执行进入放弃状态例程 Aborting，在停止状态例程 Stopping 中可编写的梯级逻辑如图 16-24 所示。

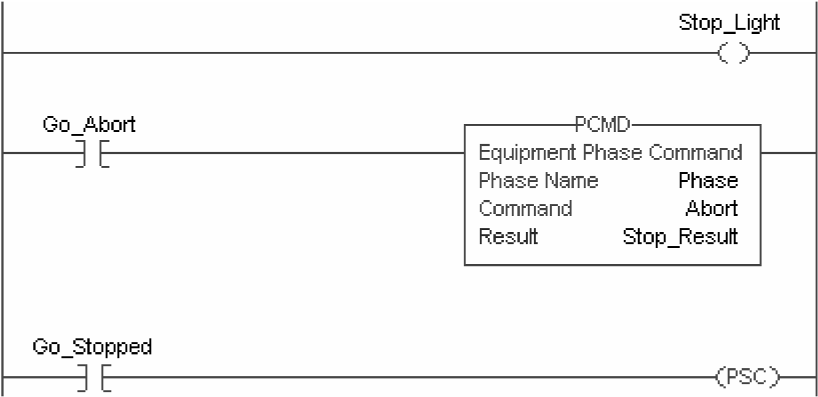


图 16-24　在例程 Stopping 中执行放弃命令的梯级逻辑

Go_Stopped 的梯级条件执行 PSC 指令，离开停止状态例程 Stopping，进入预状态 Stopped。无条件梯级控制的停止状态显示灯 Stop_Light 在离开停止状态例程 Stoppting 时，由于例程静止作用会自然熄灭。Stop_Phase 梯级条件执行的 Abort 命令将进入放弃状态例程 Aborting 进行处理。

放弃状态例程 Aborting 是非正常关机，通常是不得已的关停，这时要处理的事务比正常停机要复杂，重要的是要进行安全停车的处理。在放弃状态例程 Aborting 可编写的梯级逻辑如图 16-25 所示。

Go_Aborted 的梯级条件执行 PSC 指令，离开放弃状态例程 Aborting，进入预状态 Aborted。无条件梯级控制的放弃状态显示灯 Abort_Light，在离开停止状态例程 Aborting 时，由于例程静止作用会自然熄灭。

当一切处理完毕，要回到正常工作状态时，跟 Complete 预状态一样，Stopped 和 Aborted 的预状态也是通过 Reset 的命令执行，进入准备状态例程 Resetting，从而进入正常工作进程循环。

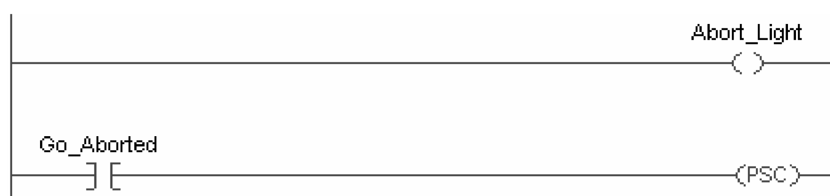


图 16-25　在例程 Aborting 中编写的梯级逻辑

虽然我们是按照学习的过程逐步增加命令指令的，但是最后发现大多命令指令都集中编写在预状态例程中，如图 16-26 所示。预状态例程是一个非常特殊的例程，类似于程序中的主控例程，总是处在被扫描状态，编写在例程中的命令指令一定会得到及时的执行，其执行动作覆盖了整个 Phase Manager。

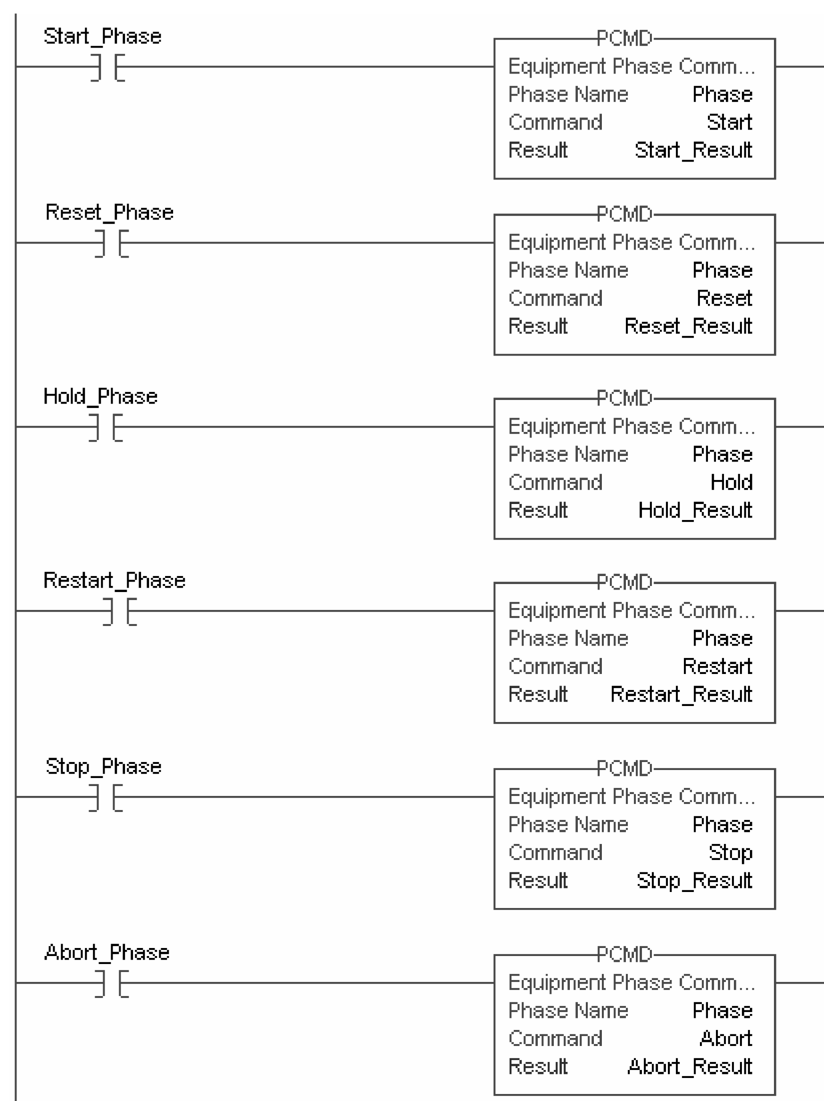


图 16-26　大多命令指令都集中编写在预状态例程中

以上，我们完成了正常生产工艺过程周期的循环、暂停的循环以及正常停机和非正常停机的循环。除了生产工艺处理过程的逻辑我们用闪烁的灯替代，基本的编程过程都经历了。你会发现，这里没有太多编程技巧可言，你惟一可做的是遵循状态模块的进程，按照规定的指令编写。如此标准的模式，即使是让一个新手来编程，也不容易出错，并能在较短的时间内完成。作为维护人员，读程序也有固定的思路和方式，而无须揣摩编程特点和熟悉程序结构，很快就能上手。这就是标准化带来的好处。

16.3 PhaseManager 的故障管理和调试编程

下面我们再讨论故障记录的编程。Equipment Phase 创建时，会产生一个同名的结构数据标签，这个结构数据标签存放了这个管理过程的许多信息，其中有一个存放故障代码的子元素，如图 16-27 所示。



图 16-27　存放故障代码的子元素

编写梯级逻辑，如图 16-28 所示。当故障源触发，相应梯级条件成立，执行 PFL 指令，将故障代码放入故障代码单元中存放，如图 16-29 所示。

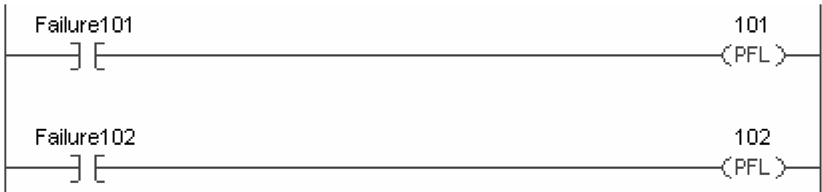


图 16-28　探测故障的梯级逻辑



图 16-29　存放故障代码的数据单元

请注意，Phase. Failure 这个单元存放的信息并不是常规的数据标签子元素，按照惯例用 CLR 指令或 MOV 指令是不能清除的，必须采用专门的清除指令 PCLF 来清除，编写梯形逻辑如图 16-30 所示。

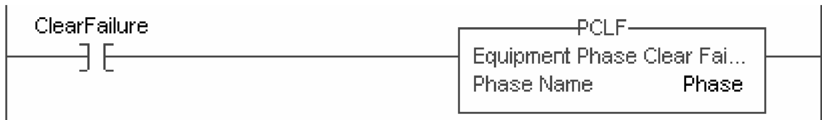


图 16-30　清除故障代码的梯形逻辑

还有一种可能会改变 Phase. Failure 单元的存储内容，那就是新的故障出现将覆盖原有的信息。如果两个以上的故障源同时发生，数字大的优先权更高，比如 101 和 102 的故障同时

发生，保留下来的信息是 102，所以要将较为重要的故障信息定义为数字大的代号。

最后，再讨论关于调试中用到的指令编程，PPD 与 AutoPause、Pausing 和 Paused 命令联合起来使用，只有在 Pausing 亚状态才作用于测试的 PPD 指令，用来中断本梯级指令的执行。Pausing 亚状态在状态模块上可观察到，这里不作更为深入的讨论。梯级逻辑编程如图 16-31 所示。

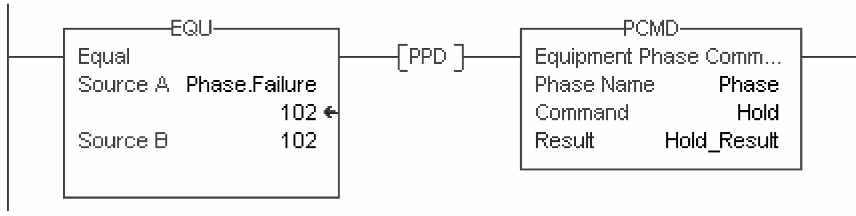


图 16-31　用 PPD 测试的梯级逻辑

PPD 指令不可以编写在预状态例程中，这是用来调试状态例程的，只能编写在状态例程中。编程限制在任何时候只能有一个 PPD 中断，并限制 PPD 为单次扫描，多次扫描将使得 PPD 连续多次作用于同一个中断点。

一个 Equipment Phase 只对应一个设备的控制，一个控制系统的多个设备要创建多个 Equipment Phase，由于生产工艺过程的设备运动是相互关联的，多个 Equipment Phase 之间也有必然的联系，这时往往需要一个充当调度者的程序来协调工作。关于多个 Equipment Phase 各自的拥有权利，也是需要编程来解决的，所以设备阶段管理还有一些指令用来实现拥有权的管理，我们这里就不作讨论了。如果你有兴趣或有需求，可阅读 Phase Manager 的设计手册，它将告诉你如何来规划设备的控制。

以上我们讨论的都是程序控制进程的设备管理，其过程如图 16-32 所示的状态模块所示，我们编程或调试，都是在观察状态模块，以了解当前哪个状态例程处在激活状态。

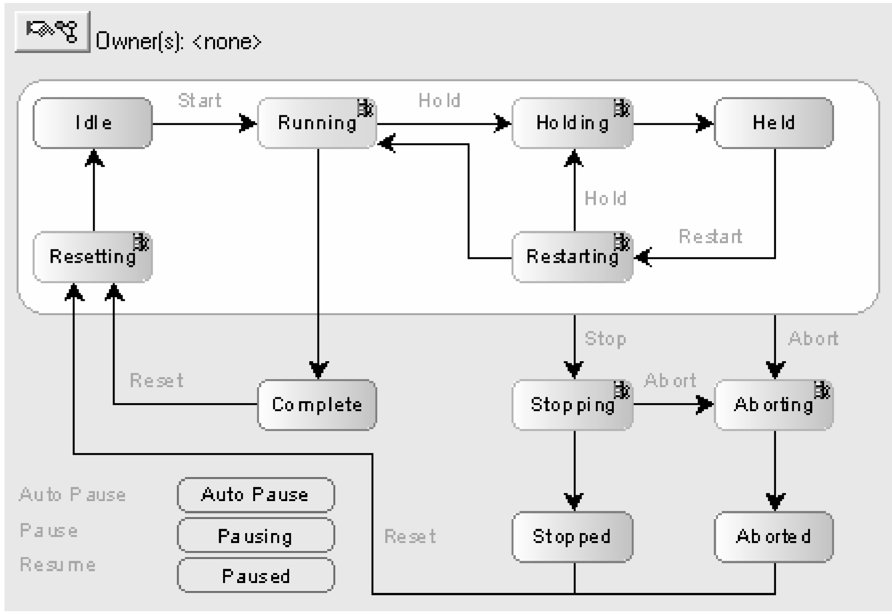


图 16-32　状态模块

在调试时，还可以手动控制状态进程，在状态模块上，选择拥有者为 RSLogix5000，即本地手动控制，如图 16-33 所示，可以看到手动操作的命令被激活，由原先的灰显转为实显，鼠标点击当前激活命令项便可进行操作。

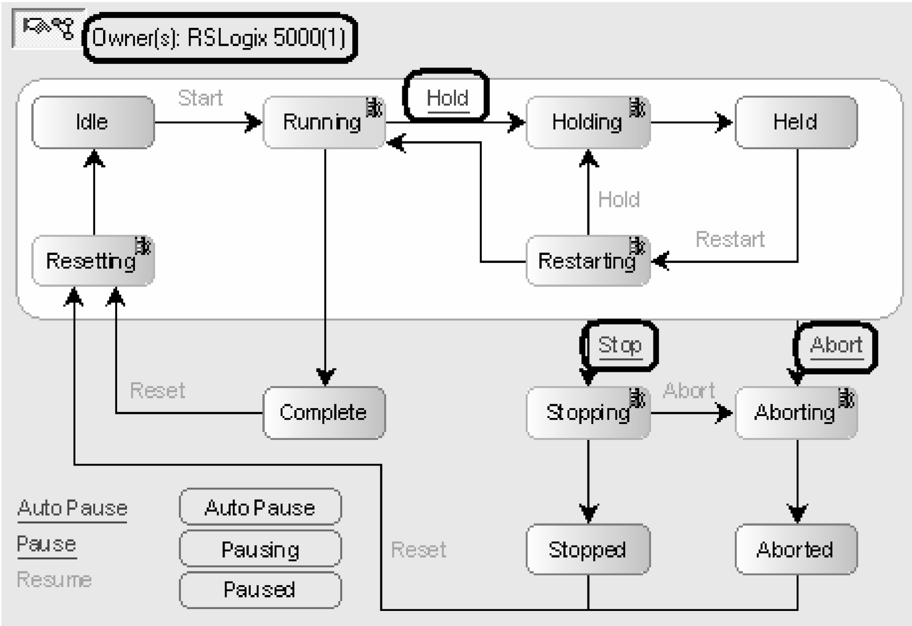


图 16-33　选择本地手动控制

第 17 章

顺序功能流程图编程介绍

顺序功能流程图（Sequential Function Chart，SFC）是顺应工程师开发项目的习惯，从功能方框图的形式演变过来的一种编程模式。这种功能结构式的编程模式，特别适合步骤清晰、进程明确的工艺过程，还大大减少了无效的梯级逻辑扫描，节约程序扫描时间，提高了系统的反应速度。

简单地说，SFC 创建了一个结构，结构都是由基本元素组成的，这个基本元素就是步和转换条件，如图 17-1 所示。任何一个步的离开与否都取决于转换条件是否成立，一旦结构完成，所谓编程就是编写步里面的执行动作以及它的转换条件。

SFC 的基本结构有如下几种：

- 顺序结构　基本元素顺序连接而成，上一步转换条件成立离开，进入本步。
- 选择分支　多个基本元素并列，选择其中之一执行，每个元素含有上一步判断进入的转换条件和离开本步的转换条件，上一步判断进入哪一步有优先之分。

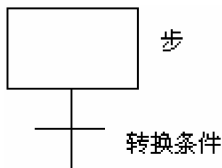


图 17-1　SFC 的基本元素

- 并行分支　多个基本元素并列，同时执行，共用上一步转换条件和离开所有并列步的转换条件，所有并列步共用的转换条件成立进入下一步。
- 循环结构　上一步转换条件成立，跳转到连接所指定的步，构成循环。

在建造好的 SFC 结构上，完成步和 Action 的组态，编写各个 Action 和转换条件：

- 步的组态　组态步的监视计时器及报警设定。
- Action 的组态和编程　安排 Action 的执行顺序和执行方式，执行方式通常采用的是 N、P1 和 P0。
- 转换条件的编程　以不同的方式提供一个 BOOL 量状态，从而决定转换条件成立与否。

下面我们用一个简单的例子来完成一个 SFC 的控制过程。

编程满足如下要求：

- Step1：Light1 闪烁 15s　　Condition1：用步的计时完成位离开；
  Step2：Light2 闪烁 15s　　Condition2：用普通计时器定时离开；
  Step3：Light3 闪烁 15s　　Condition3：用子例程调用离开；
  Step4：Light4 闪烁　　Condition4：用手动按钮操作离开；
  Step5：Counter 加 1　　Condition5：判断条件离开。
- 每进行一次以上步的循环，Counter 计数加 1，当步循环 3 次之后，进入 SFC 停止

待命状态，完成一次生产过程；

- 用 SFR 指令回到 SFC 初始步，SFR 指令操作使用手动按钮；
- 在初始步编制有关的初始化逻辑，清除步循环计数，为生产循环次数加 1，从初始步进入步循环动作使用外部手动按钮操作；
- 输出点的闪烁可引用用户自定义指令 FLASH 来执行，要求步后扫描时复位闪烁的点。

17.1 建立 SFC 的结构和步的编程

根据需求画出的框图结构如图 17-2 所示。每个步骤和它的转换条件，最后是判断分支，当步循环小于 3 时，转向 Step1 执行，当步循环大于等于 3 时，进入停止待命。初始化步的进入，取决于外部控制，要编程解决从停止待命进入初始化步。

下面按照功能框图来创建 SFC 的结构图，并编写相应的步、Action 和转换条件，以及相关的梯级逻辑。

创建名为 SFC_Test 的例程，从 4 种编程类型中选择 SFC 的类型，如图 17-3 所示。

打开 SFC_Test 例程，如图 17-4 所示，出现的是默认的初始步，以双线框表示。不要改变这个特定的双线框步，在它前面添加新的步和转换条件，否则当控制器进入初始化步的操作时，会找不到系统所认定的初始化步。总之，保留这个双线框的步作为初始化步。

关于 Initial 步的编程

将 Step_000 改名为 Initial，此为初始化步。选中 Initial 步，点击右键后，点击“Add Action”，如图 17-5 所示。

在初始化步 Initial 上增加两个 Action，Action_000 用来清除步循环计数器，Action_001 用来累加生产循环次数，Initial 步用来完成进入生产循环的初始化工作。选择执行方式为 P1，即步激活时执行一次便不再执行。初始化步的转换条件由命名为 Start_SFC 的 BOOL 量执行，这个标签可以别名给外部开关输入点，一旦按钮操作启动，便可从初始化步进入步循环的工作。增加的两个 Action 如图 17-6 所示。

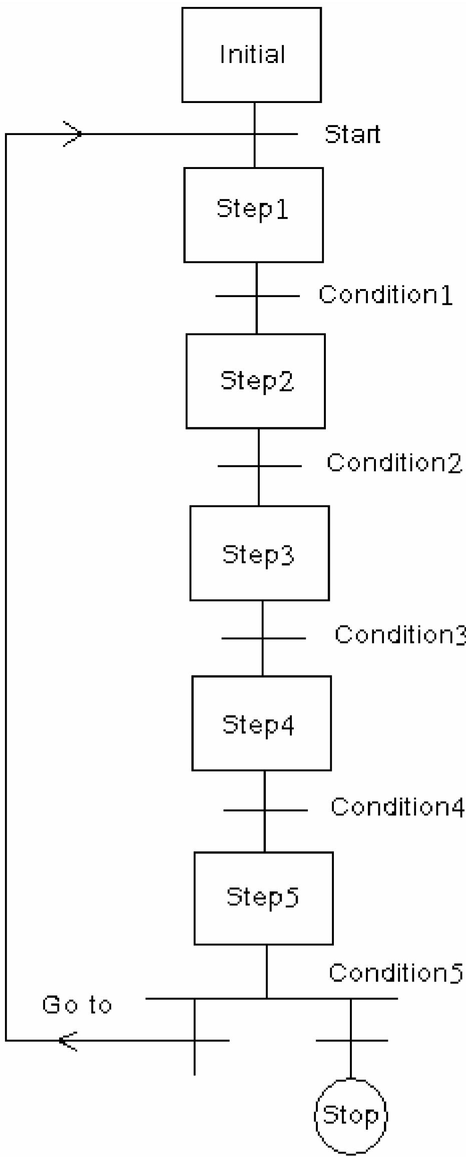


图 17-2 框图结构

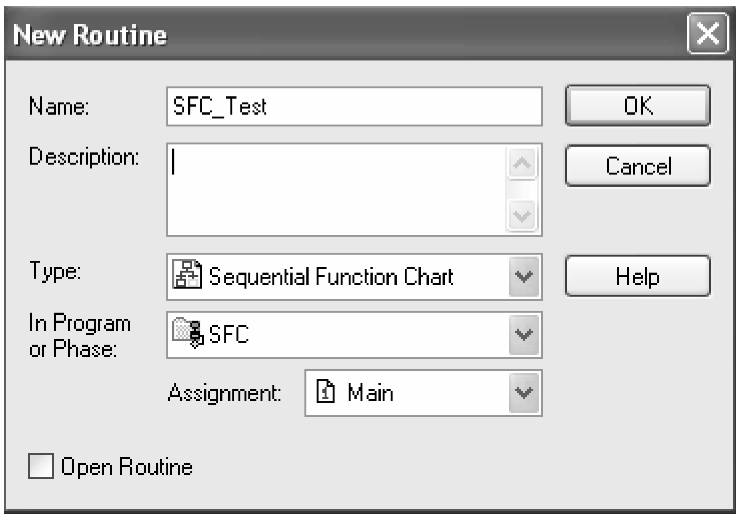


图 17-3　创建名为 SFC_Test 的例程

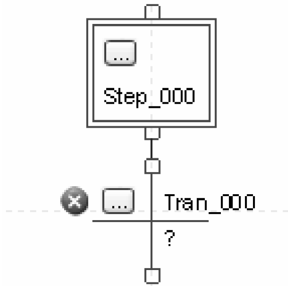


图 17-4　SFC_Test 例程

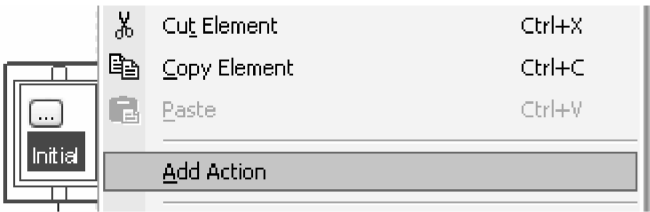


图 17-5　创建 Action

Action_000 和 Action_001 的执行动作采用语句编程代码的赋值语句完成，这是保持型赋值，即使离开这个步，数值依然保持。Action_000 赋值 0 给标签 Counter，这是清除步计数器。根据需求，当步循环完成 3 次，SFC_Test 会离开步循环，重新开始新的一轮生产循环，进入新的一轮生产循环前需要清除原来的完成状态，这便是初始化的工作。Ac-

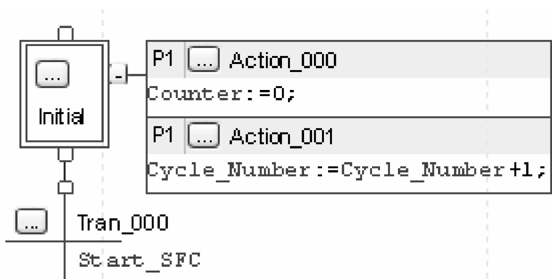


图 17-6　初始化步

tion_001 作为累加器，生产循环周期计数 Cycle_Number 加 1 之后回送给自己，完成累加的工作，这是为生产循环而累积的记录。

关于 Step1 步的编程

进入 Step_001 组态，设定步计时器预置值为 15000，即 15s 后，步计时器完成位置位，如图 17-7 所示。

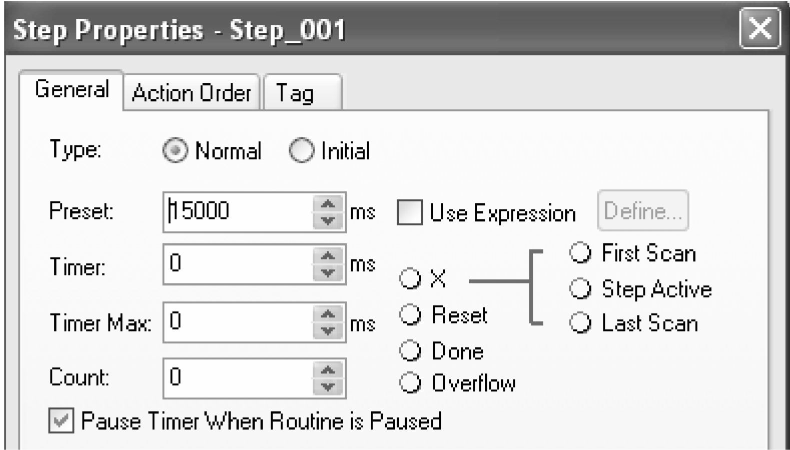


图 17-7　进入 Step_001 组态设定步计时器预置值

在步 Step_001 上增加一个 Action，选择执行方式为 N，即在步激活的所有时间都会执行。Action_002 调用例程 Step1_Routine，步转换条件定为步计时器的完成位 Step_001. DN，步激活扫描 15s 之后，步计时器完成位置位，步转换条件成立，离开步 Step_001，如图 17-8 所示。

在例程 Step1_Routine 中编写梯级逻辑，令 Light1 闪烁，如图 17-9 所示。梯级条件采用无条件，当例程调用时，FLASH 指令参数项 Out_Flash 的输出 Light1 将闪烁。

关于 Step2 步的编程

在 Step_002 上增加一个 Action，选择执行方式为 N，即在步激活的所有时间都会执行。Action_003 调用例程 Step2_Routine，步转换条件为常规计时器的完成位，例程 Step2_Routine 中的计时器计时

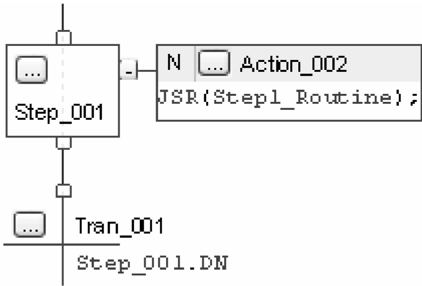


图 17-8　Action_002 调用例程 Step1_Routine

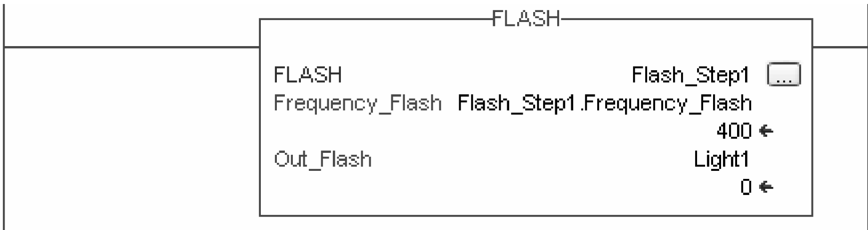


图 17-9　在例程 Step1_Routine 中编写的梯级逻辑

15s 之后，Timer_Step2. DN 完成位置位，步转换条件成立，离开 Step_002，如图 17-10 所示。

在例程 Step2_Routine 中编写梯级逻辑，令 Light2 闪烁，如图 17-11 所示。梯级条件采用无条件，当例程调用时，FLASH 指令参数项 Out_Flash 的输出 Light2 将闪烁。编写通延时计时器指令 TON，计时 15s 后完成位 Timer_Step2. DN 置位，梯级条件为无条件，步离开的后扫描将复位计时器 Timer_Step2，等下次进入这个步的时候计时器仍能正常完成计时工作。

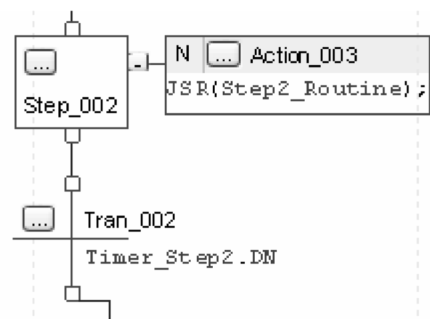


图 17-10　Action_003 调用例程 Step2_Routine

关于 Step3 步的编程

在步 Step_003 上增加一个 Action，选择执行方式为 N，即在步激活的所有时间都会执行。Action_004 调用例程 Step3_Routine，步转换条件调用例程 Step3_Timer 来完成，例程 Step3_Timer 的计时器计时 15s 之后，完成位置位，返回给步转换条件成立，离开步 Step3，如图 17-12。

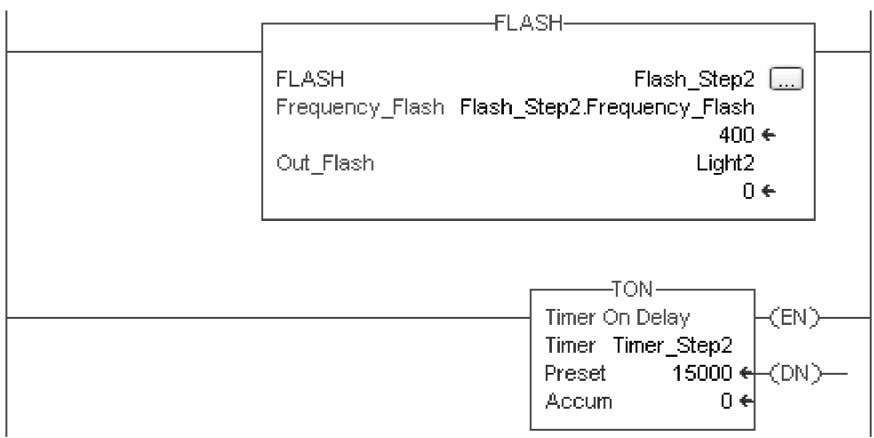


图 17-11　在例程 Step2_Routine 中编写的梯级逻辑

在例程 Step3_Routine 中编写梯级逻辑，令 Light3 闪烁，如图 17-13 所示。梯级条件采用无条件，当例程调用时，FLASH 指令参数项 Out_Flash 的输出 Light3 将闪烁。

在例程 Step3_Timer 中编写计时器计时，15s 后完成位置位，EOT 指令返回给 Step3 的步转换条件，从而离开 Step3 步，如图 17-14 所示。EOT 指令是一条专用指令，专用来返回转换条件例程中产生的转换条件的布尔量。注意，这个计时器没有机会自行复位，转换条件例程是没有后扫描的。

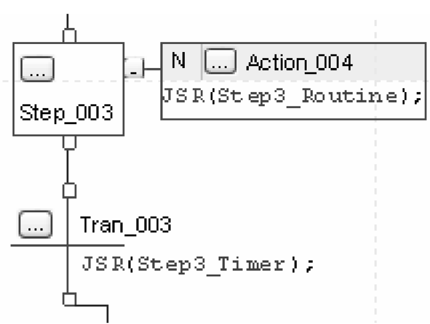


图 17-12　Action_004 调用例程 Step3_Routine

关于 Step4 步的编程

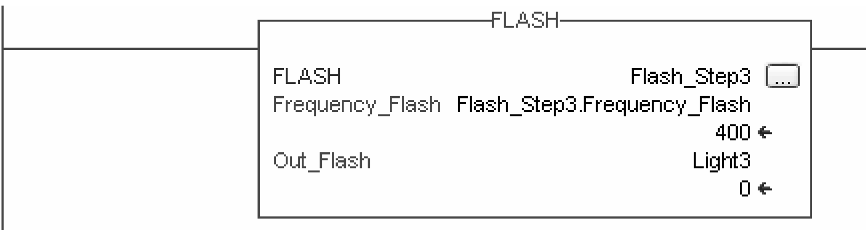


图 17-13　在例程 Step3_Routine 中编写梯级逻辑

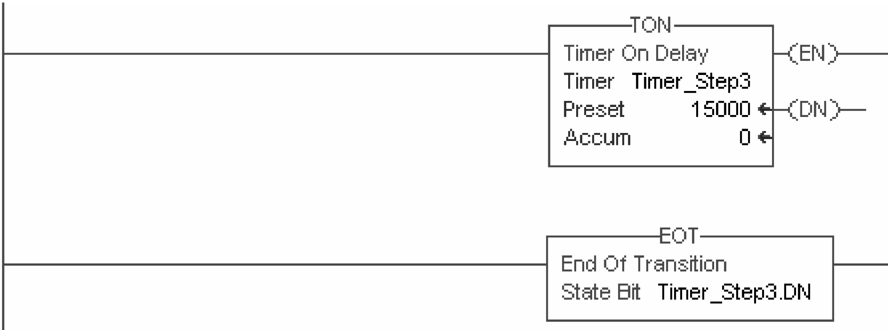


图 17-14　在例程 Step3_Timer 中编写计时器计时

在步 Step4 上增加一个 Action，选择执行方式为 N，即在步激活的所有时间都会执行。Action_005 调用例程 Step4_Routine，步转换条件用手动操作 BOOL 量 Leave_Step4 来完成，如图 17-15 所示。

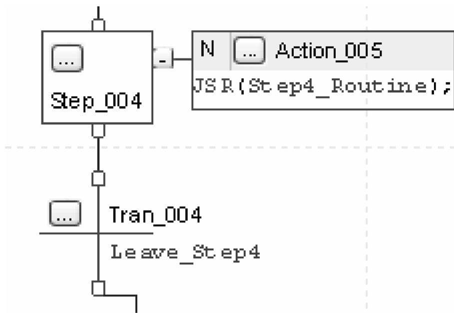


图 17-15　Action_005 调用例程 Step4_Routine

在例程 Step4_Routine 中编写梯级逻辑，令 Light4 闪烁，如图 17-16 所示。梯级条件采用无条件，当例程调用时，FLASH 指令的参数项 Out_Flash 的输出 Light4 将闪烁。

同时，Step3 步转换条件的例程没有后扫描的操作帮助复位计时器，在 Step4 中完成计时器复位。这个只是我们在模拟例程转换条件使用了计时器才遇到的问题，特殊处理而已，没有示范意义，不过请留意，转换条件例程的处

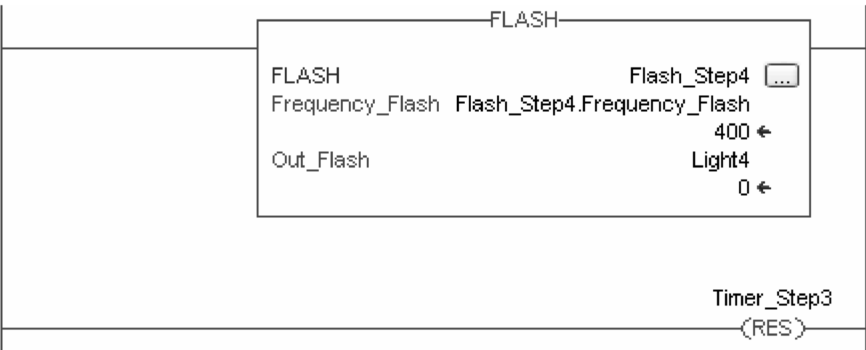


图 17-16　在例程 Step4_Routine 中编写的梯级逻辑

理和步的例程处理是不一样的，没有后扫描或预扫描的处理，系统没有机会自行处理，需要用户编程来解决，所以放在下一步来处理。

关于 Step5 步的编程

在步 Step5 上增加一个 Action，选择执行方式为 P1，即步激活时执行一次，Action_006 执行赋值语句，令计数器 Counter 加 1，步转换条件则采用了选择分支，当计数器 Counter 小于 3，跳转到步 Step1 继续执行；当计数器 Counter 大于等于 3，进入 Stop，即停止待命状态，等待 SFC 复位，进入下一轮的工作循环，如图 17-17 所示。

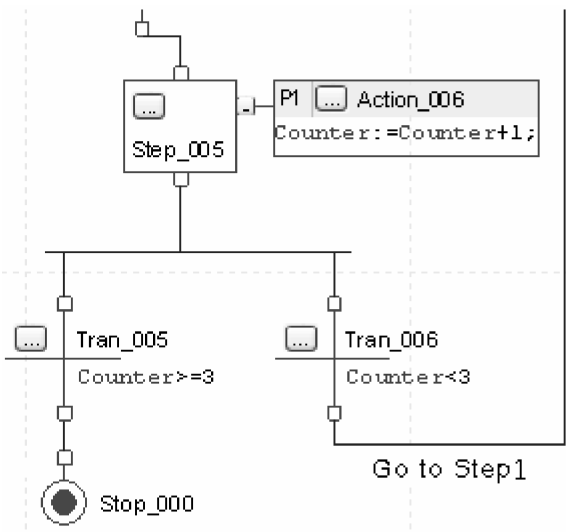


图 17-17　Action_006 实行加 1 操作

此处 Counter 的比较一定要用大于等于 3，不要用仅仅等于 3，因为等于 3 的惟一判断一旦由于意外离开 3 就不能进入 Stop，变得无所适从，这也是调试中得到的经验。

在创建 Step、Action 和 Tran 的时候，你将发现系统会自动命名，这是因为编程软件的设置所致。在编程软件 Options 项中可以找到如图 17-18 所示的设置页面。系统自动命名是可以更改成用户所指定的名称的，数据库中的标签名也将随之更改。

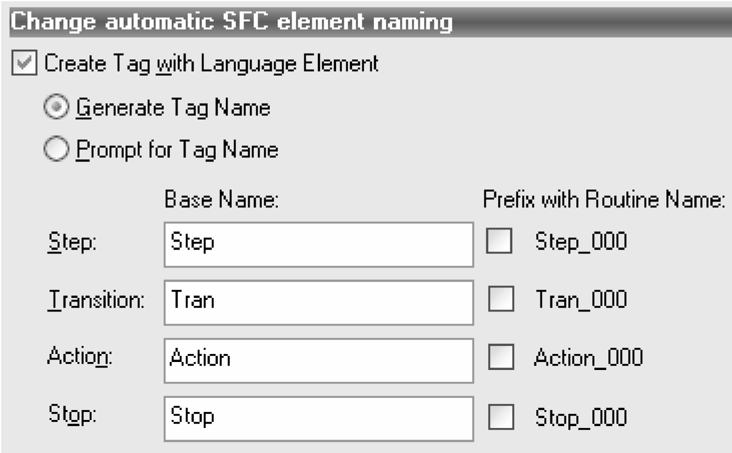


图 17-18　标签命名规则设置页面

至此，一个简单的 SFC 过程就完成了，我们特地采用了不同的方式离开步，以体验不同处理方式的编程。如图 17-19 所示是一个完整的 SFC 流程图。

为了 SFC 测试过程能够完整地运行下来，还需要增加另外的操作，这就是 SFC 例程之外的操作管理。

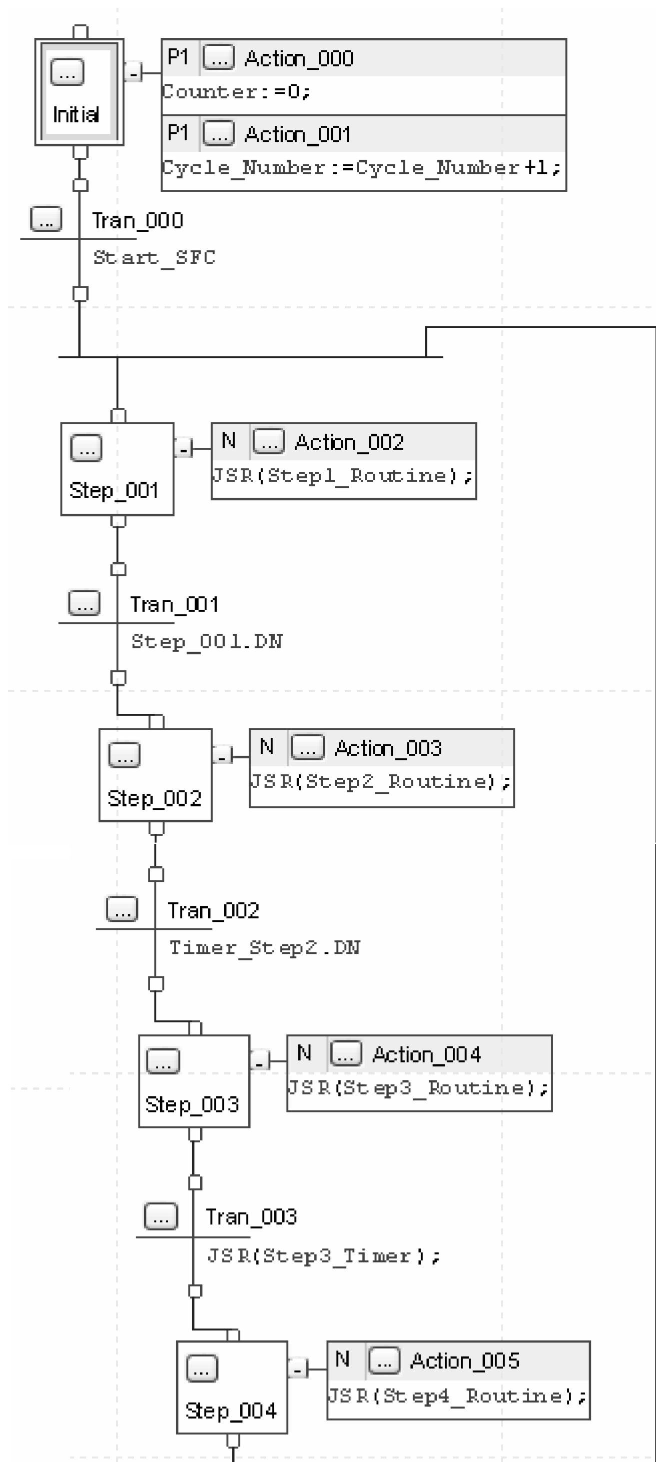


图 17-19 完整的 SFC 顺序功能流程图

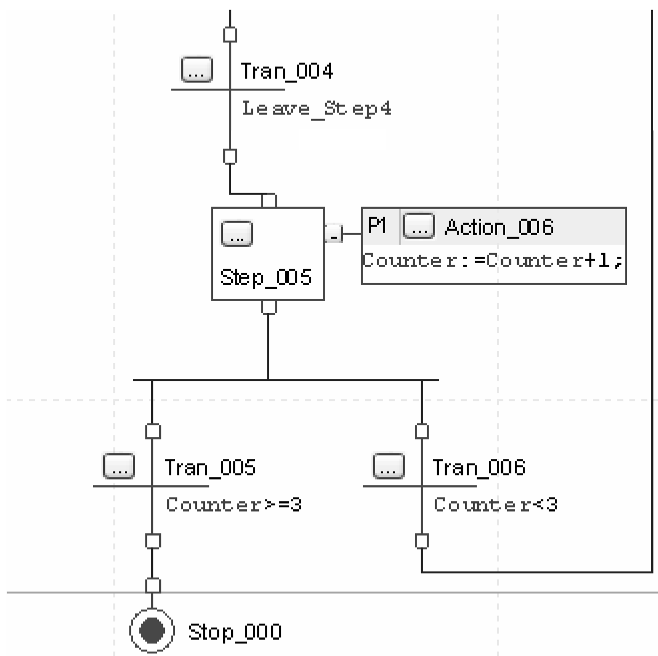


图 17-19　完整的 SFC 顺序功能流程图（续）

17.2　SFC 例程运行的外部操作管理

创建例程 MainRoutine，并设为主控例程，名为 SFC 程序下的例程包含了前面在编程步的时候创建的相关的例程，如图 17-20 所示。

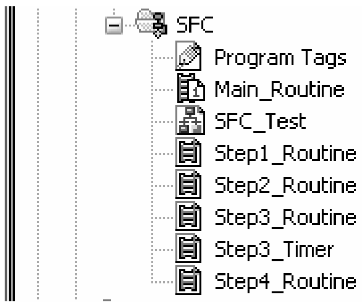


图 17-20　SFC 程序下的例程

在主例程 Main_Routine 中，编写调用例程 SFC_Test 的梯级逻辑，如图 17-21 所示。这是一个无条件调用，SFC_Test 的例程将保持连续扫描。

将项目下载到控制器运行，测试的结果完全正常。但是你有没有疑惑，离开步的时候，这个闪烁的灯会不会有时是熄灭的，有时仍然亮着。你应该没有忘记，当时我们测试 Add_On 指令就发生了这样的情形，后来我们在扫描模式中编写了梯级条件不成立的处理，解决了这个问题。正是这个梯级条件不成立的逻辑处理帮助我们解决了离开步的善后工作，因为离开步的后扫描令所有的梯级条件不成立，我们得到了想要的梯级条件不成立的结果。如果我们没有在梯级条件不成立的例程中做相应的处理，或者我们希望离开步时具有另外的处理方式，则可以

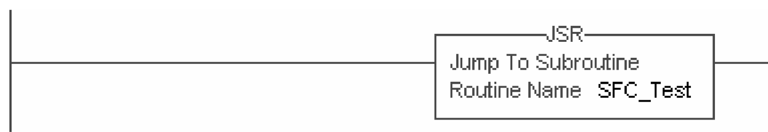


图 17-21　调用例程 SFC_Test 的梯级逻辑

回到 Add_On 指令进行修改，在后扫描的例程中编写某些处理。

让我们回到 Add_On 指令的扫描模式的组态页面，增加后扫描例程，如图 17-22 所示。

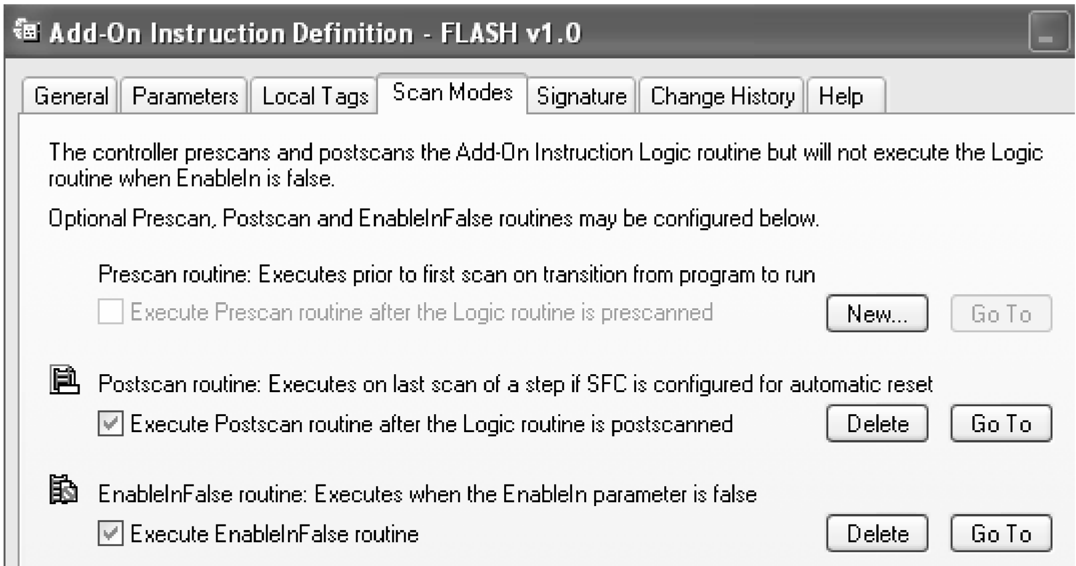


图 17-22　在扫描模式的组态页面上增加后扫描例程

在 Add_On 指令的 FLASH 下面新增加了 Postscan 例程，作为后扫描的处理，如图 17-23 所示。

在 Postscan 例程中编写梯级逻辑，如图 17-24 所示，复位输出的位状态。当例程经历后扫描时，将关闭这个不确定的状态位。当然这个实例中梯级条件不成立的处理已经做了这个工作，这里似乎是重复了。重要的是，我们关注了后扫描在离开步时指令是如何表现的，一旦我们需要后扫描的不同处理时，就知道在什么地方，以什么样的方式去解决。

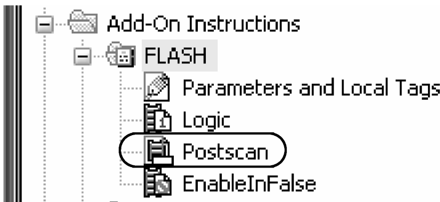


图 17-23　在 Add_On 指令的 FLASH 下面新增加了 Postscan 例程

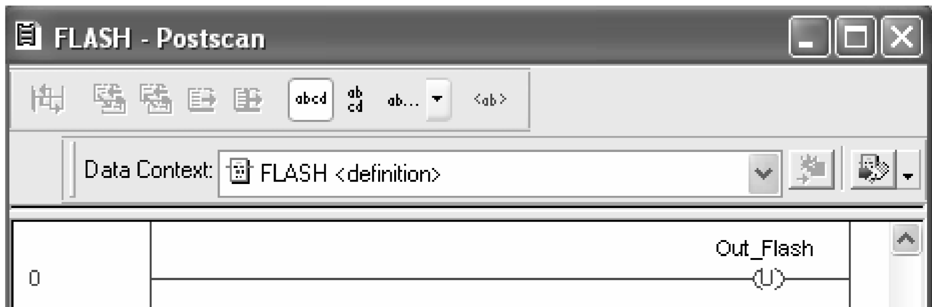


图 17-24　在 Postscan 例程中编写的梯级逻辑

与此同时，全局的 SFC 执行组态也要进行相应的设置，在控制器属性的 SFC Execution 页面设置后扫描为 Automatic reset，注意默认设置是 Don't scan，如果此处不改变设置的话，

刚才在 Add_On 指令里面编写的后扫描处理例程是不会起作用的。改变设置后的选择如图 17-25 所示。

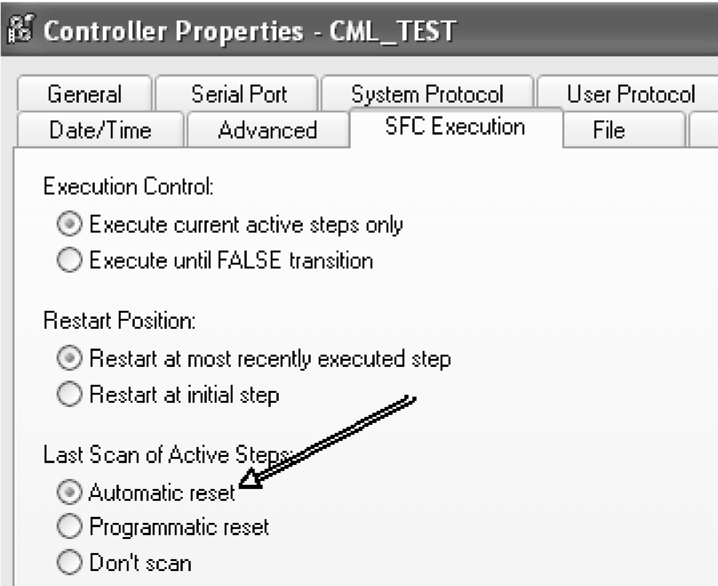


图 17-25　改变设置后的选择

现在可以进行完整的调试了，控制器运行，进入初始化步 Initial ，分配手动操作别名给外部开关进行相应的操作，当步的循环完成 3 次之后，进入到停止待命，如图 17-26 所示，可以看到意味着结束的停机待命的 Stop_000 此时是激活的状态。

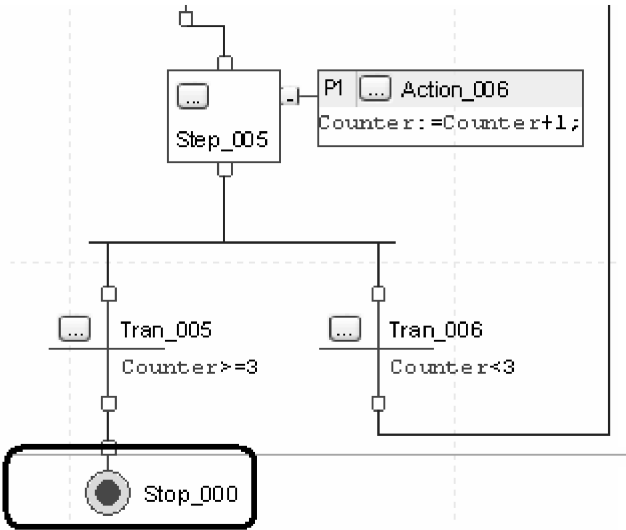


图 17-26　处于停机待命状态

如果要开始一个新的生产循环，Stop 步应该转入 SFC_Test 的初始化步，这个可以用 SFC 的复位指令 SFR 来操作，这条指令应该编写在 SFC 之外的例程中，并且被持续扫描。在主例程中，我们再添加一个梯级逻辑，如图 17-27 所示。

用复位操作 Reset_SFC_Test 位来复位 SFC_Test，停机状态位 Stop_000. X 在 Stop 激活

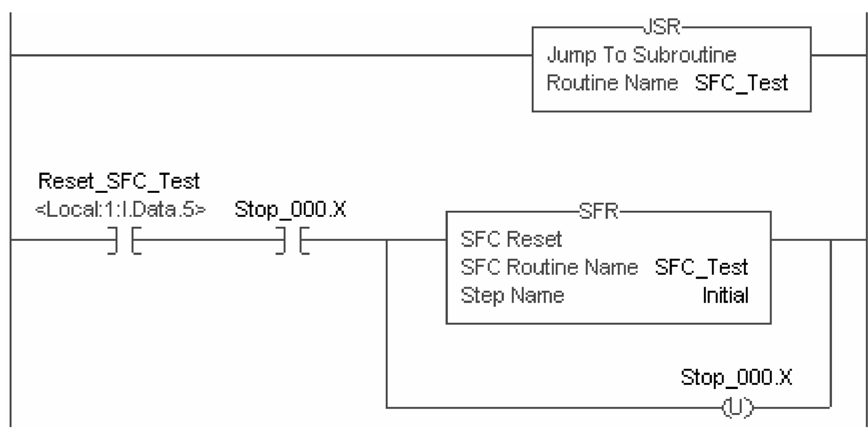


图 17-27 添加复位的梯级逻辑

状态的时候是置位的，复位 SFC_Test 的同时解锁这个位，作为梯级条件的 Stop_000. X 被解锁后，令梯级条件消失，这样确保了复位操作只会执行一次。

复位操作 Reset_SFC_Test 可别名给外部开关点，实现外部手动操作。

刚才，我们用一个简单的例子完成了 SFC 的全过程，虽然简单，却经历了 SFC 所有的步骤，即使面对一个真实的应用需求，所实行的也是这样的一个过程，只是在步的 Action 中有更为复杂的过程处理，或者分支有更多的变化而已。

第 18 章

功能块编程简要介绍

结构数据类型标签将大量不同类型的基本数据集合在一起，这正是 DCS 系统的数据结构特征，于是 DCS 系统使用的功能块轻而易举地移植到了 PAC 中，从而给我们带来功能块运用的便利，在某些场合下，能明显感受到功能块的功能优势。

如果说梯形图编辑的梯级逻辑是编程的话，在 DCS 系统功能块的编辑则称为组态，功能块将特定的功能集于一体，不必编写很多执行逻辑，只要根据应用需求选择适合的功能块，设定相关的参数便可以了。Logix 控制器用到的功能块，既有简单功能的功能块，其作用与梯形图指令近似，也有专用功能的功能块，功能强大而参数复杂。本章只对功能块编程进行一般的介绍，仅对两类功能块举例说明，而不详细介绍控制器每个功能块的运用。

18.1　累加器功能块组态

在梯形图编程中，我们曾为流量的累加算法绞尽了脑汁，而在功能块的组态中却可直接使用，功能块组态看上去似乎是简单的，但是要弄明白为数不少的参数，却不是一件轻松的事情。下面，我们试着了解累加器功能块 TOT 的基本运用。如图 18-1 所示的是累加器的功能块面板。

单是看功能块的外部接点，参数已是不少，这仅仅是在功能块外部连接所显示的部分，点击功能块 TOT 右上角的[...]，展开功能块参数表，我们看到的参数更多。如图 18-2 所示的是功能块 TOT 输入参数的一部分，外部显示的 Vis 选项被勾选，该参数将作为外接的输入连接点，跟提供输入参数的其他功能块的输出端连接，或者与输入参数连接符相连接。

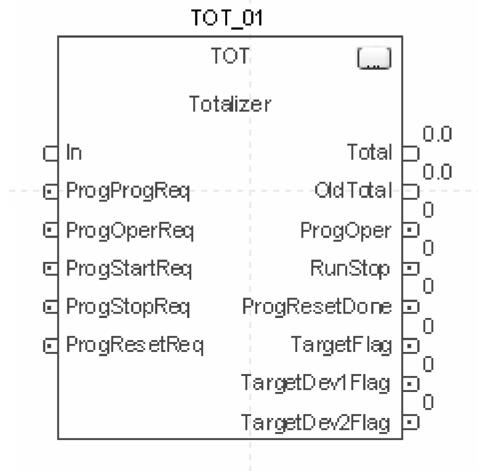
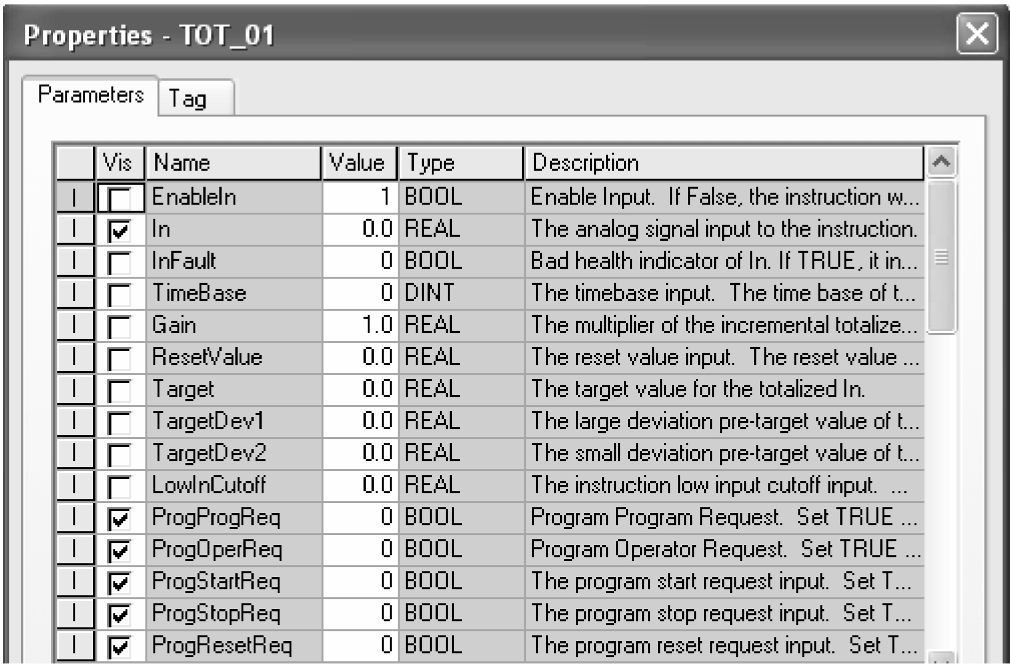


图 18-1　累加器的功能块面板

每个参数的简要说明同时显示在参数的注释中，不但有参数的基本定义，还有数据的设置方式和含义，一般情况无须查找手册便可明了。说明如下：

- EnableIn　功能块输入使能，类似于梯形图指令的梯级条件，默认值设置为 1，功能块被使能执行。
- In　采集模拟量信号的数据输入，此为



	Vis	Name	Value	Type	Description
I	☐	EnableIn	1	BOOL	Enable Input. If False, the instruction w...
I	☑	In	0.0	REAL	The analog signal input to the instruction.
I	☐	InFault	0	BOOL	Bad health indicator of In. If TRUE, it in...
I	☐	TimeBase	0	DINT	The timebase input. The time base of t...
I	☐	Gain	1.0	REAL	The multiplier of the incremental totalize...
I	☐	ResetValue	0.0	REAL	The reset value input. The reset value ...
I	☐	Target	0.0	REAL	The target value for the totalized In.
I	☐	TargetDev1	0.0	REAL	The large deviation pre-target value of t...
I	☐	TargetDev2	0.0	REAL	The small deviation pre-target value of t...
I	☐	LowInCutoff	0.0	REAL	The instruction low input cutoff input. ...
I	☑	ProgProgReq	0	BOOL	Program Program Request. Set TRUE ...
I	☑	ProgOperReq	0	BOOL	Program Operator Request. Set TRUE ...
I	☑	ProgStartReq	0	BOOL	The program start request input. Set T...
I	☑	ProgStopReq	0	BOOL	The program stop request input. Set T...
I	☑	ProgResetReq	0	BOOL	The program reset request input. Set T...

图 18-2　功能块 TOT 输入参数的一部分

累加量的增值。

- InFault　采集模拟量信号有误，当此状态标识置位，TOT 将不累加和刷新数据。
- TimeBase　累加器的采集时间基值，设置为 0，时基为秒；设置为 1，时基为分钟；设置为 2，时基为小时。采用枚举量设置，只有符合枚举量的才是有效数据，无效时基将不会累加刷新数据，并给出错误状态。
- Gain　累加量增加的比值，相乘输入值后获得设定放大倍数的累积增值，默认值为 1，即不进行放大。
- ResetValue　复位累加量，只有当程序复位请求或操作员复位请求为真时复位操作才起作用。
- Target　累加量的目标值，设定累加量目标值。
- TargetDev1　累加量目标值的大偏差值，设定离目标值较大的差值。
- TargetDev2　累加量目标值的小偏差值，设定离目标值较小的差值。
- LowInCutoff　输入关闭门槛值，当输入值低于该值时，累计器停止累加操作。
- ProgProgReq　用户程序请求程序控制，由请求程序控制的用户程序设置为真，保持此项为真，在操作员控制模式下不起作用，一旦操作员模式为假，则锁定程序控制模式。
- ProgOperReq　用户程序请求操作员控制，由请求操作员控制的用户程序设置为真，保持此项为真，可锁定操作员控制模式。
- ProgStartReq　程序启动请求输入，由请求启动的用户程序设置为真。
- ProgStopReq　程序停止请求输入，由请求停止的用户程序设置为真。
- ProgResetReq　程序复位请求输入，由请求复位的用户程序设置为真。

如图 18-3 所示的是功能块 TOT 输出参数的一部分，外部显示的 Vis 选项被勾选，该参数将作为外接连接点，为其他功能块的输入端连接点提供输入数据，或者与输出参数连接符相连接。

Properties - TOT_01

Parameters　Tag

	Vis	Name	Value	Type	Description
0	☐	EnableOut	0	BOOL	Enable Output.
0	☑	Total	0.0	REAL	The totalized value of the In. Arithmetic...
0	☑	OldTotal	0.0	REAL	The value of the total before a reset oc...
0	☑	ProgOper	0	BOOL	Program/Operator control indicator. TR...
0	☑	RunStop	0	BOOL	The indicator of the operational state of...
0	☑	ProgResetDone	0	BOOL	The indicator the Totalizer has complet...
0	☑	TargetFlag	0	BOOL	The flag for Total greater than or equal ...
0	☑	TargetDev1Flag	0	BOOL	The flag for TargetDev1. Set TRUE w...
0	☑	TargetDev2Flag	0	BOOL	The flag for TargetDev2. Set TRUE w...
0	☐	LowInCutoffFlag	0	BOOL	The instruction low input cutoff flag out...
0	☐	DeltaT	0.0	REAL	Elapsed time between updates.
0	☐	Status	16#...	DINT	Bit mapped status of the function block.
0	☐	InstructFault	0	BOOL	Instruction has generated a fault

图 18-3　功能块 TOT 输出参数的一部分

功能块 TOT 部分输出参数的说明：

- EnableOut　使能输出，显示功能块处于使能状态，功能块被执行。
- Total　累加器当前累加量值，按照设定的采样时间模式进行累积。
- OldTotal　复位前的累加量值，访问该项，可获得复位时准确的累加值。
- ProgOper　程序/操作员模式显示，程序模式为真，操作员模式为假。
- RunStop　累加器操作状态显示，累加器运行为真，累加器停止为假。
- ProgResetDone　累加器程序模式复位请求完成显示，用户程序可以用此位探测复位成功，当程序模式复位请求为假时，该位转为假。
- TargetFlag　累加量目标值标志位，当累加值达到累加器目标值时为真。当该位置位后，累加值仍然继续累积，此处只表明了到达的标志，并不影响累积的操作。
- TargetDev1Flag　累加量目标值的大偏差值标志位，当累加量离目标值还差大偏差值时为真。
- TargetDev2Flag　累加量目标值的小偏差值标志位，当累加量离目标值还差小偏差值时为真。
- LowInCutoffFlag　输入关闭门槛值标志位，当输入小于等于关闭门槛值时，该位为真。
- DeltaT　刷新数据之间所花费的时间，一般跟采样时间一致。
- Status　功能块位映射状态。
- InstructFault　功能块故障显示位，功能块故障发生为真。

见识过功能块参数之后，我们来讨论功能块之间的连接方式。所谓功能块组态，一个最

基本的工作就是将各个功能块连接起来。

功能块的组态是连接功能块连接点的连线，前一个功能块的输出端连接点连接到下一个功能块的输入端连接点，由此构成功能块的信息流。功能块连接点之间的连接有两种方式：一种是用连接线连接，一种是用连接符连接。当功能块相互挨近时可用连接线直接连接，当要连接的功能块连接点之间相距较远，需要跨越其他功能块形成交叉线时，宜用连接符连接。用同名标志表示它们的连接关系，或者直接指定连接的标签名称。

功能块的输入端和输出端也就是功能块的外部参数连接点，点击功能块参数提供的外部连接点，用连接线连接。数据量连接点之间的连接线为实线，离散量连接点之间的连接线为虚线。

在功能块的连接点上，有两种不同的连接标识来连接不同数据类型的连接线：

- 离散量连接点标识：连接的功能块参数为布尔量，相应的连接线为虚线。
- 数据量连接点标识：连接的功能块参数为实数或双整数，相应的连接线为虚线。

功能块组态的连接符连接方式，可从指令选取框左边的 4 个连接符中选择所需要的连接符号，放置并与相应的连接点连接。

- 输入参数连接符：可选取本程序数据库和全局数据库中的任何布尔数、实数和双整数，也可直接键入立即数。
- 输出参数连接符：可选取本程序数据库和全局数据库中的任何布尔数、实数和双整数。
- 连线接入连接符：连接同一例程中的较远位置的连接点，选取与连线接出点相同的标志符号。
- 连线接出连接符：连接同一例程中的较远位置的连接点，建立与连线接入点对应的标志符号。

功能块之间的组态连接实际上建立了功能块例程的执行结构和执行顺序，每条连线都是从一个功能块的输出到下一个功能块的输入，从而构成了信息的流向，成为功能块执行的顺序。在多个功能块连接关系并行的情形下，不容易看出执行的顺序，每个功能块的执行顺序在未经校验的功能块例程中显示为未知，只有在校验功能块例程之后，本功能块的执行顺序才会出现在功能块页面的左下角，当增减功能块之后，再次校验功能块例程，将重新排列每个功能块新的执行顺序。

在程序下创建功能块例程，如图 18-4 所示。

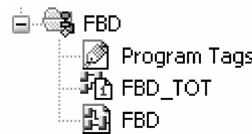


图 18-4　在程序下创建功能块例程

我们在例程 FBD_TOT 中组态一个累加器 TOT 功能块，如图 18-5 所示。这里只有单独的一个功能块，该功能块中，通过参数连接符连接来自于梯形图控制的标签，作为功能块的输入参数，其输出参数通过参数连接符传递到控制器的其他标签。

现在我们将模拟量通道定标后的数据 Analog_L 接入累加器功能块的输入，并设定时基为 0，即采集时间基值为秒，设定比例系数为 1，累加量目标值为 30000，最大偏差值为 3000，最小偏差值为 1000，最低门槛值为 50，参数设置画面如图 18-6

所示。

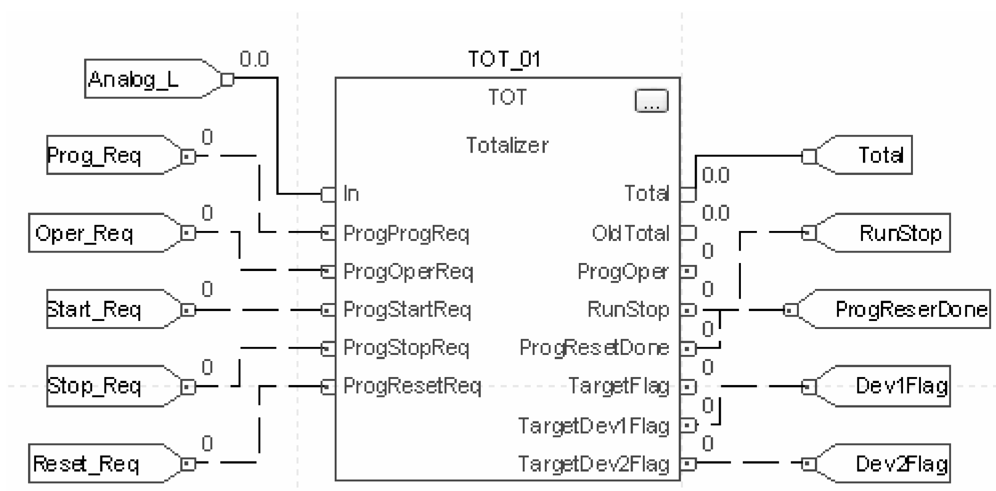


图 18-5　在例程 FBD_TOT 中组态一个累加器 TOT 功能块

Properties - TOT_01

Parameters | Tag

	Vis	Name	Value	Type	Description
I	☐	EnableIn	1	BOOL	Enable Input. If False...
I	☑	In	143.86641	REAL	The analog signal inp...
I	☐	InFault	0	BOOL	Bad health indicator o...
I	☐	TimeBase	0	DINT	The timebase input. ...
I	☐	Gain	1.0	REAL	The multiplier of the in...
I	☐	ResetValue	0.0	REAL	The reset value input....
I	☐	Target	30000.0	REAL	The target value for t...
I	☐	TargetDev1	3000.0	REAL	The large deviation pr...
I	☐	TargetDev2	1000.0	REAL	The small deviation pr...
I	☐	LowInCutoff	50.0	REAL	The instruction low in...
I	☑	ProgProgReq	1	BOOL	Program Program Re...
I	☑	ProgOperReq	0	BOOL	Program Operator Re...
I	☑	ProgStartReq	1	BOOL	The program start req...

图 18-6　功能块 TOT_01 的参数设置画面

在梯形图中编写相应的梯级逻辑作为操作控制，如图 18-7 所示。通过梯级逻辑的控制可以改变 TOT 的运行模式、启动和停止累加操作以及复位累加值。我们不需要对累加的过程编写任何相关的操作，只需对 TOT 的参数进行设置和控制就可以了，整个组态过程非常简单快速。

功能块执行的情况如图 18-8 所示。可以看出当 Prog_Req 设置为 1，且 Start_Req 设置为 1 时，累加器工作被启动，累加开始，当累加值达到 27368. 3 时 ，刚刚越过离目标值 30000 的大偏差 3000，也即 27000 的界限，此时大偏差值标志位 TargetDev1Flag 置位；尚未达到离

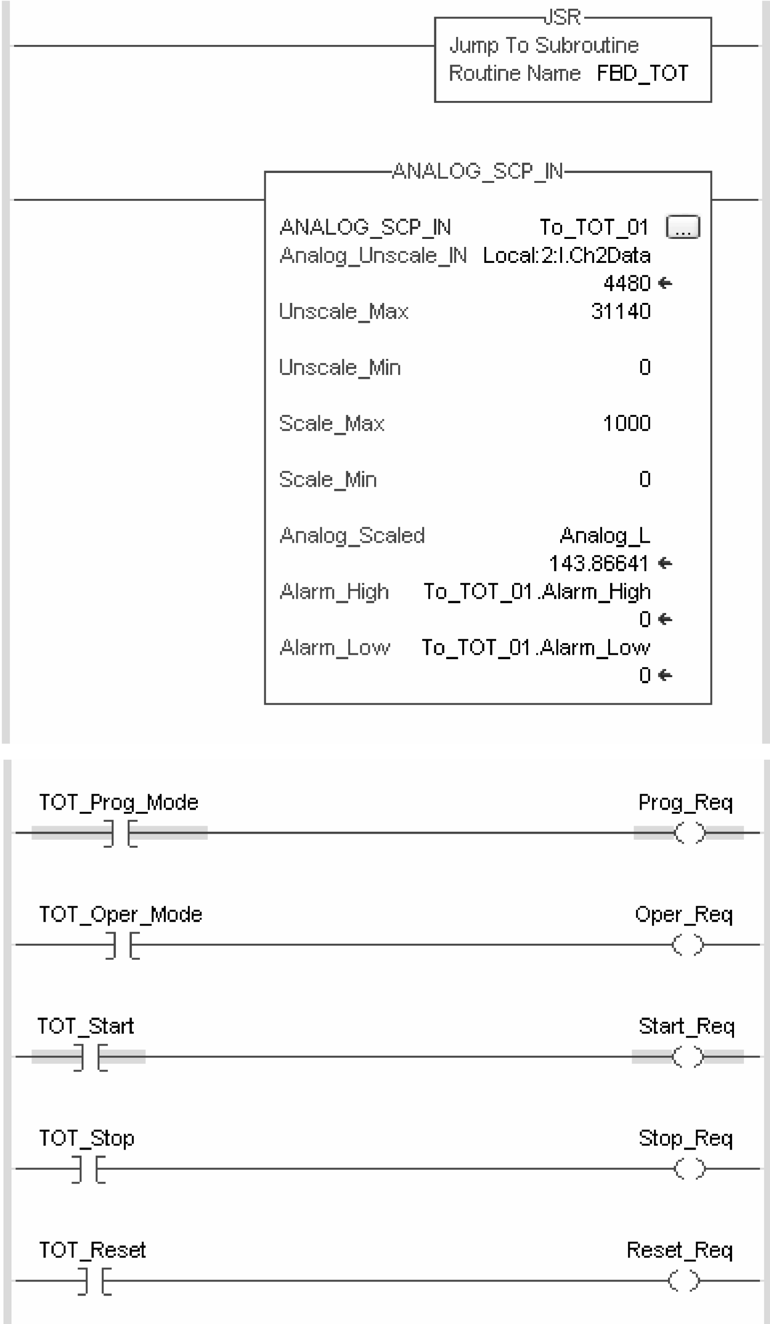


图 18-7 在梯形图中编写相应的梯级逻辑作为操作控制

目标值 30000 的小偏差 1000，也即 29000 的界限，此时小偏差值标志位 TargetDev2Flag 复位。稍加留意可以看到，TOT 功能块上次是在累加值等于 51223. 836 时进行了复位。状态标志位 RunStop 为 1 标识了 TOT 正在进行累加工作。

在这里，我们可以享受功能直接使用的乐趣，而不需要费尽心机地去解决可能遇到的数据处理的麻烦。毫无疑问，调试也是非常的方便和快捷。但是我们要花费时间去了解功能块

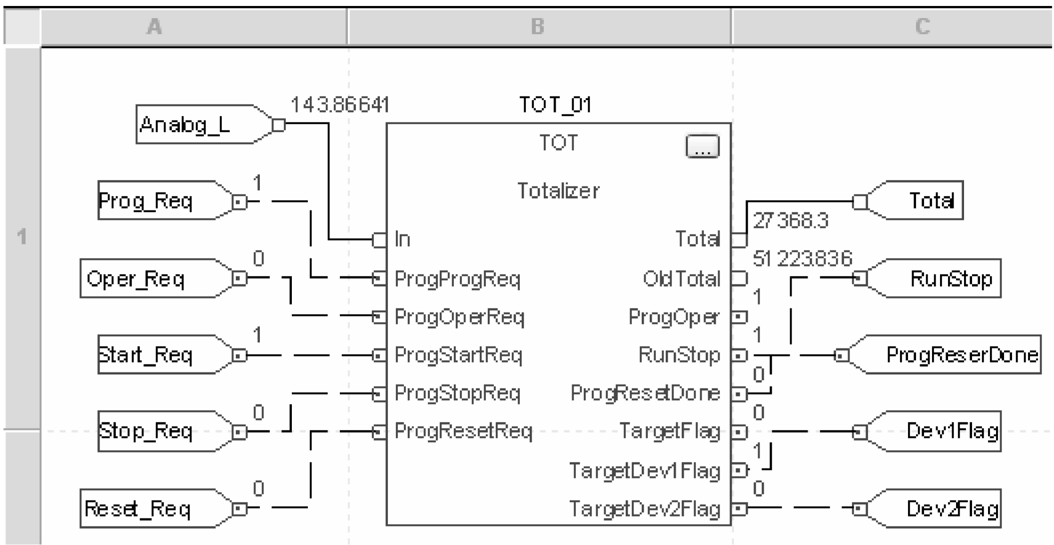


图 18-8　功能块执行的情况

的参数，以便能准确地运用，如果功能块使用过程出现问题，一定是参数使用不当所致。

功能块 PIDE 也是功能强大的指令，凸显了功能块的优势。其 PID 调节性能十分优越，如果你针对同一对象，采用梯形图的 PID 指令调试过，然后换成 PIDE 来调试，一定会感觉到明显的调试性能的差异。如果你购买的编程软件含有功能块编程，我建议你采用功能块 PIDE 来调试闭环控制。关于功能块 PIDE 的组态，我们这里就不再介绍了，无非是大量的参数意义说明，它的外部接线正如我们熟知的那样，构成了闭环控制结构。

18.2　简单功能块的综合运用

还有一部分功能块跟梯形图的指令功能是相同或相似的，用起来有时也很方便。下面，我们试着用一些功能块来完成熟悉的梯形图指令功能。

首先，我们看看计数器在功能块中的用法。功能块的增减计数集成在一个功能块中，由增计数使能端 CUEnable 的跳变引起累积数加 1；减计数使能端 CDEnable 的跳变引起累积数减 1。在梯形图逻辑编程中，我们不得不通过技巧来解决，采用一个梯级的 CTU 指令和一个梯级的 CTD 指令来共同修改同一个计数器结构数据标签，每个梯级则用各自的计数梯级条件，即增计数脉冲和减计数脉冲。在功能块中，使用一个计数器便可以了，如图 18-9 所示。

此外，可以看到，计数器功能块的预置值也可以在输入端直接用立即数给定，不一定要通过一个双整字标签来设定，或进入参数表中直接设定。

梯形图中的宽脉冲变窄脉冲的前沿触发脉冲和后沿触发脉冲指令，用功能块也能实现，组态非常简单，功能块的组态如图 18-10 所示。对于功能块来说，就是输入宽脉冲，输出窄脉冲。

与梯形图指令相似的简单功能块，我们也可用来解决类似梯形图的需求。下面的功能块实例编程是采用与梯形图相似的指令功能，我们来对比运用。

加法功能块如图 18-11 所示。加数 Analog1 和被加数 Analog2 作为源数据，提供给功能

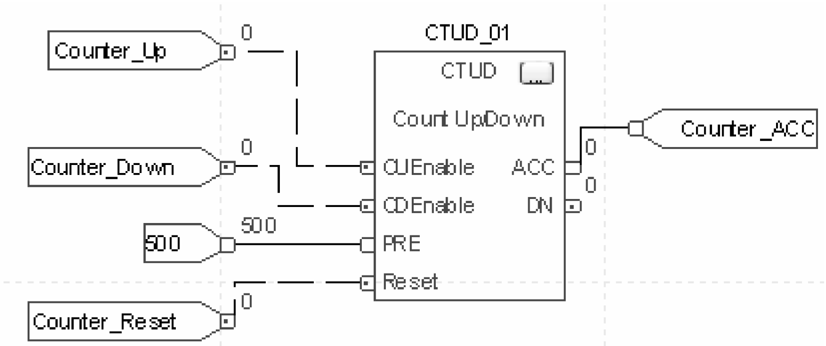


图 18-9　具有增减计数的计数功能块

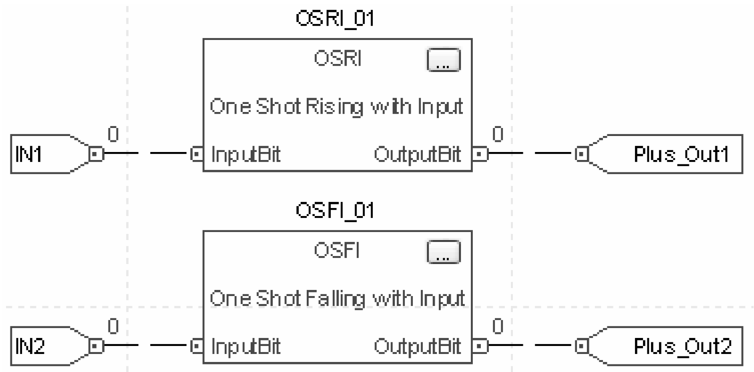


图 18-10　前沿触发和后沿触发的功能块

块 ADD 的输入端，功能块相加之后的和作为目标地址输出引向功能块 ADD 的输出端。

减法功能块如图 18-12 所示。减数 Analog1 和被减数 Analog2 作为源数据，提供给功能块 SUB 的输入端，功能块相减之后的差作为目标地址输出引向功能块 SUB 的输出端。

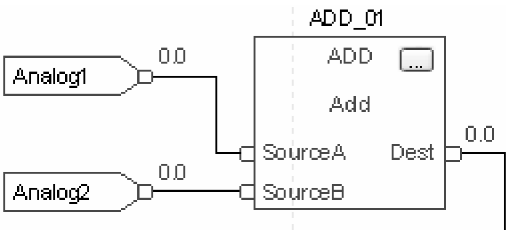


图 18-11　加法功能块

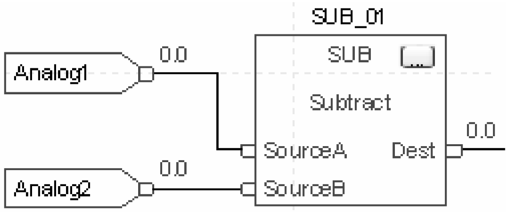


图 18-12　减法功能块

将相加之后的和与相减之后的差送入选择功能块 SEL，如图 18-13 所示。加法的和被送入功能块 SEL 的输入端 In1，减法的差被送入功能块 SEL 的输入端 In2，选择开关由输入点 Select_IN 的状态决定，可由外部或程序控制，为 0 时选择输出为加法的和，为 1 时选择输出为减法的差。

功能块 SEL 的参数设置如图 18-14 所示。默认是功能块使能，即 EnableIn 为 1，例程被扫描时功能块将被执行。如果希望控制功能块的执行或不执行，可以用程序来控制这个使能位。

将功能块 SEL 的输出送入功能块 HLL 的输入，如图 18-15 所示。HLL 是上限设定报警和下限设定报警的功能块，根据设定的上限值和下限值，输入数据根据判断将产生相应的报警位作为输出端提供给需要的功能块或者标签值。

功能块 HLL 的参数设定如图 18-16 所示。当输入值 In 大于等于上限值 900 时，上限报警位 HighAlarm 设置为 1；当输入值 In 小于等于下限值 100 时，下限报警位 LowAlarm 设置为 1。当输入值高于上限值或低于下限值时，输出被限幅在上限值或下限值。

如果我们希望上限值报警位和下限值报警位点亮的报警灯闪烁，

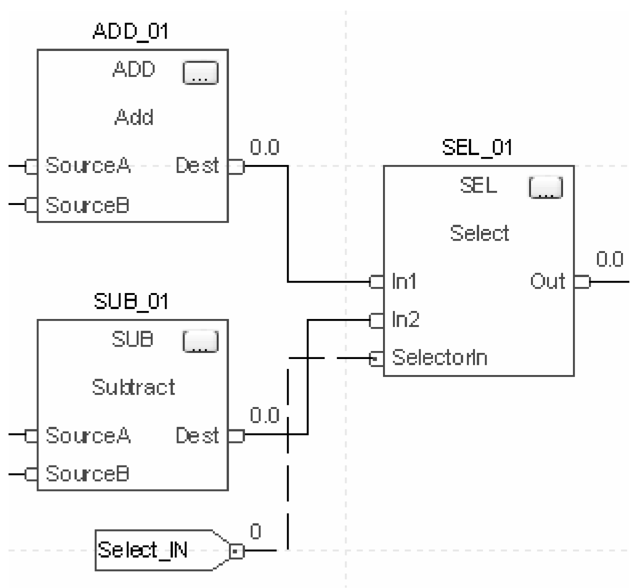
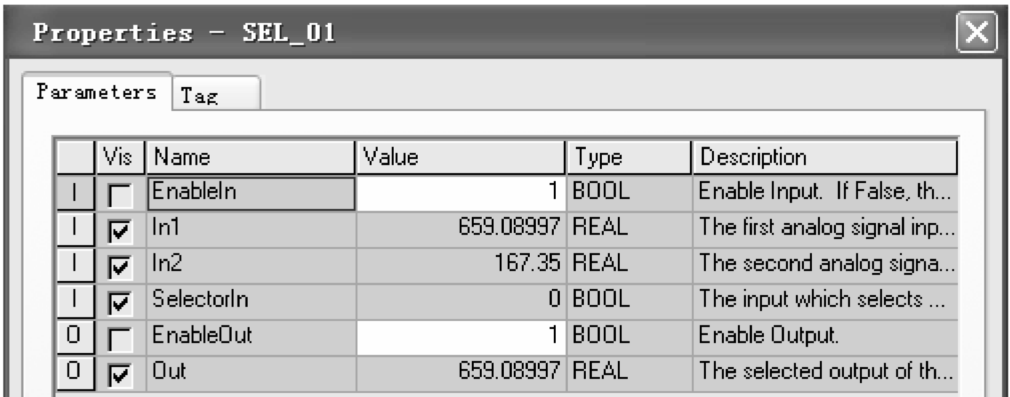


图 18-13　将相加之后的和与相减之后的差送入选择功能块 SEL


Properties - SEL_01

Parameters　Tag

	Vis	Name	Value	Type	Description
I	☐	EnableIn	1	BOOL	Enable Input. If False, th...
I	☑	In1	659.08997	REAL	The first analog signal inp...
I	☑	In2	167.35	REAL	The second analog signa...
I	☑	SelectorIn	0	BOOL	The input which selects ...
O	☐	EnableOut	1	BOOL	Enable Output.
O	☑	Out	659.08997	REAL	The selected output of th...

图 18-14　功能块 SEL_01 的参数设置

可以增加一个计时器来实现，选取一个可复位计时器的 TONR 功能块，如图 18-17 所示。选取输入点 Enable_TONR 作为计时器使能位，可由外部或程序控制，计时器 TONR_01 完成位 DN 直接连接至复位输入端 Reset，计时器 ACC 值输出引至下一个功能块。当可复位计时器如此连接便成为一个自复位的计时器。

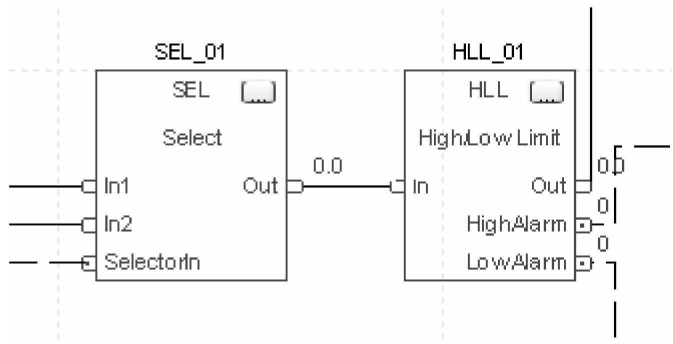


图 18-15　将功能块 SEL 的输出送入功能块 HLL 的输入

Properties - HLL_01

Parameters | Tag

	Vis	Name	Value	Type	Description
I	☐	EnableIn	1	BOOL	Enable Input. If False, th...
I	☑	In	659.08997	REAL	The analog signal input t...
I	☐	HighLimit	900.0	REAL	The high limit for the Input.
I	☐	LowLimit	100.0	REAL	The low limit for the Input.
I	☐	SelectLimit	0	DINT	SelectLimit input to the in...
O	☐	EnableOut	1	BOOL	Enable Output.
O	☑	Out	659.08997	REAL	The output of the H/L Li...
O	☑	HighAlarm	0	BOOL	The high alarm indicator ...
O	☑	LowAlarm	0	BOOL	The low alarm indicator o...
O	☐	Status	16#0000_0000	DINT	Bit mapped status of the f...
O	☐	InstructFault	0	BOOL	Instruction generated a f...
O	☐	LimitsInv	0	BOOL	HighLimit <= LowLimit
O	☐	SelectLimitInv	0	BOOL	Value of SelectLimit is N...

图 18-16　功能块 HLL_01 的参数设定

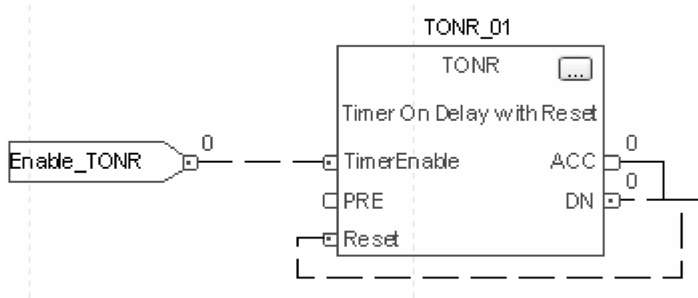


图 18-17　可复位计时器 TONR 功能块

计时器功能块参数设置如图 18-18 所示。将计时器预置值设为 400ms，人体视觉残留时间是 200ms，闪亮和熄灭各 200ms 将令人感觉闪烁。计时器后面的功能块将使用 400ms 来作为闪亮和熄灭的周期时间。

将计时器的 ACC 输出接至大于比较功能块 GRT 的输入端，如图 18-19 所示。功能块 GRT 输入是计时器的 ACC 值，输出值为 BOOL 量，当比较条件满足时，输出置为 1；比较条件不满足时，输出置为 0。

功能块 GRT 的参数设置如图 18-20 所示。比较对象是 200，当输入小于等于 200 时，输出为 0；当输入大于 200 时，输出为 1。前面的计时器预置值是 400，所以一半的时间输出为 1，一半的时间为 0，这个输出是一个闪烁的输出。

将上面的功能块 GRT 的输出并联在两个与门上，如图 18-21 所示。这两个与门分别与高报警位和低报警位相连。这样，在与门的控制下，高报警输出 HighAlarm_OUT 和低报警输出 LowAlarm_OUT 的灯便会闪烁，输出值 Anaglog_OUT 的输出范围在 100 ~ 900 之间。

完整的功能块顺序图如图 18-22 所示。在整个过程中，我们很少编辑逻辑关系，而是在

Properties - TONR_01

Parameters | Tag

	Vis	Name	Value	Type	Description
I	☐	EnableIn	1	BOOL	Enable Input. If False, th...
I	☑	TimerEnable	0	BOOL	Timer Enable. Enable tim...
I	☑	PRE	400	DINT	PRE. Timer Preset Value...
I	☑	Reset	0	BOOL	Reset. Request to reset ...
O	☐	EnableOut	1	BOOL	Enable Output.
O	☑	ACC	0	DINT	Accumulated time in millis...
O	☐	EN	0	BOOL	EN. Timer Enabled outp...
O	☐	TT	0	BOOL	TT. Timer Timing output....
O	☑	DN	0	BOOL	DN. Timing Done output...
O	☐	Status	16#0000_0000	DINT	Bit mapped status of the f...
O	☐	InstructFault	0	BOOL	Instruction has generated...
O	☐	PresetInv	0	BOOL	Preset invalid.

图 18-18　计时器功能块 TONR_01 的参数设置

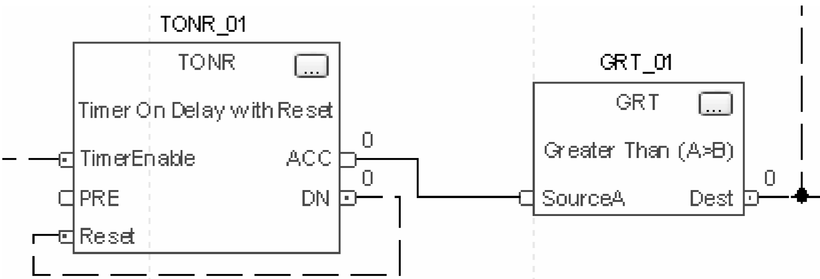


图 18-19　将计时器的 ACC 输出接至大于比较功能块 GRT 的输入

Properties - GRT_01

Parameters | Tag

	Vis	Name	Value	Type	Description
I	☐	EnableIn	0	BOOL	Enable Input. If False, the instruction will ...
I	☑	SourceA	0.0	REAL	Source A value
I	☐	SourceB	200.0	REAL	Source B value
O	☐	EnableOut	0	BOOL	Enable Output.
O	☑	Dest	0	BOOL	Result of compare block

图 18-20　功能块 GRT_01 的参数设置

选择合适的功能块，在功能块中设定合适的参数，并选择合适的对外连接，这就是功能块的组态。

这样编辑完成的功能块并没有确定它们的执行顺序，只有在 FBD 例程通过校验之后，点开其中一个功能块参数页面，查看左下角，才可以看到这个功能块的执行顺序。这里点开的是功能块 SEL 的页面，可以看到这个功能块的执行顺序是 4，如图 18-23 所示。

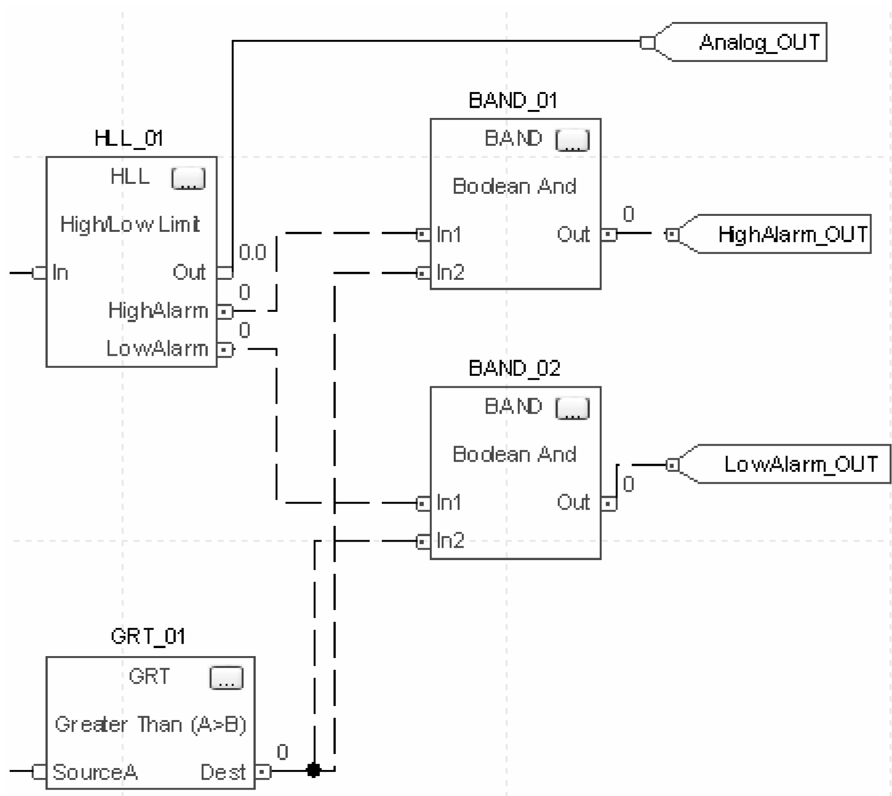


图 18-21　比较功能块 GRT_01 和高报警位/低报警位通过与门相连

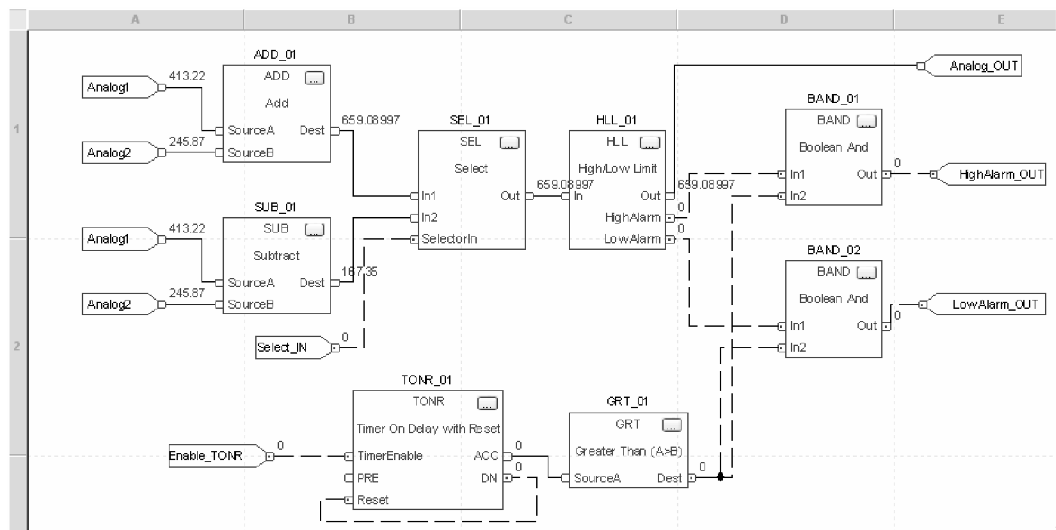


图 18-22　完整的功能块顺序图

功能块组态更偏重于特定功能性的处理，是功能性很强的指令，但是运用不够灵活，诸如数组类的处理，则不如梯形图指令方便，尤其是类似硬件的寄存器操作，更是梯级逻辑处理的特权。在 PAC 控制器中组合梯形图和功能块的特长来运用，这是 Logix 控制器的产品优势之一。

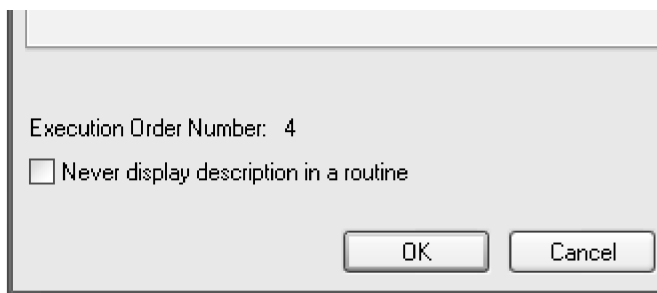


图 18-23　功能块 SEL 页面可看到执行顺序

另外，功能块没有对外通信的操作功能，诸如 MSG 指令的操作大多由梯形图逻辑和语句结构编程来担任，梯级逻辑编写 MSG 是最方便和直观的，尤其是指令的组态。

第 19 章

语句结构编程简要介绍

编写在例程中的执行代码，除了前面所提到的梯形图编程方式、SFC 编程方式和功能块编程方式，还有第四种编程方式，即语句结构编程方式。语句结构编程采用 ASCII 码字符编写执行代码，是一种基层代码编写的模式，所以语句结构编程具有很大的适应性。编程人员喜欢在顺序功能流程图（SFC）中引用语句结构编程，在步或转换条件中来表达简单的判断关系，或直接为数据标签赋值。如图 19-1 所示是一个在 SFC 例程中使用语句结构的实例。

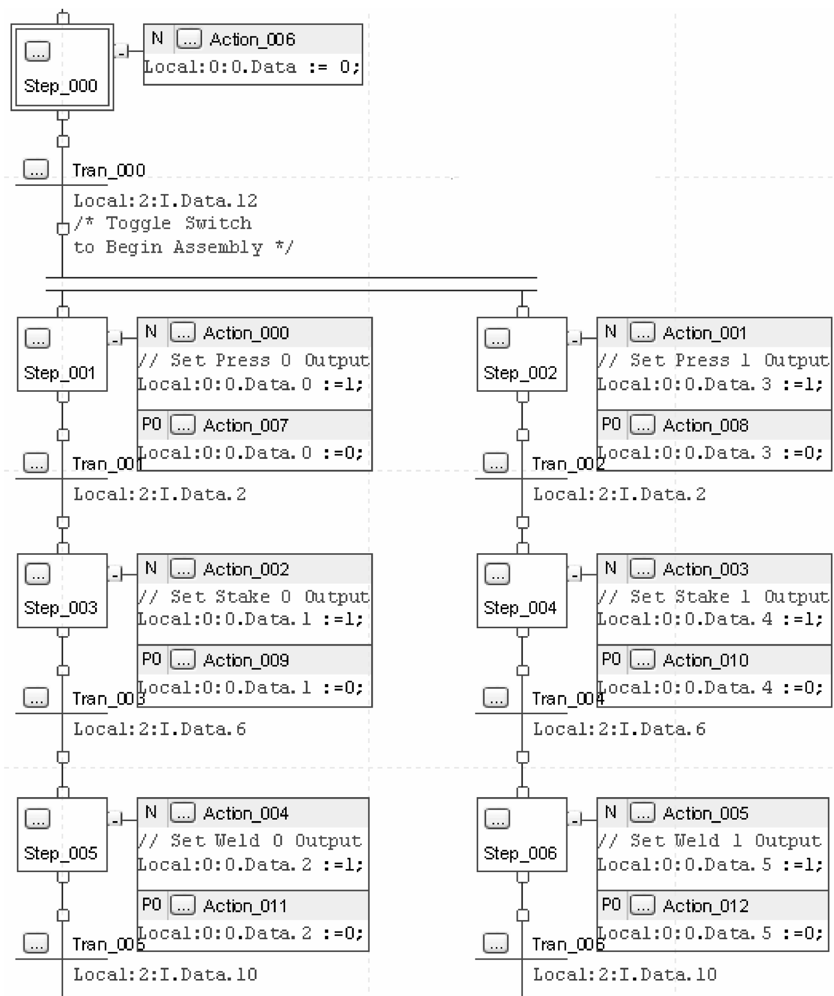


图 19-1 在 SFC 例程中使用语句的实例

语句编程类别并不复杂，可分为赋值语句、结构语句和指令语句，但使用灵活多变。本章就常用的几个实例来了解语句编程的运用。

19.1　常见赋值语句和结构语句的编程

用语句编写的代码也可以实现梯级逻辑或功能块的等同功能，最常见的语句是 IF THEN 结构语句。作为判断执行，IF 所判断的对象常常是一个逻辑量，所以可能是一个布尔量标签，也可能是一个比较表达式，判断其逻辑结果是否成立，然后决定成立时执行什么样的动作，不成立时执行什么样的动作。请看如图 19-2 所示的这个例子。

```
IF  conveyor_direction  THEN
     Light := 0;
ELSE
     Light [:=] 1;
END_IF;
```

图 19-2　IF THEN 例子

这里，传输带方向标签 conveyor _ direction 和灯标签 Light 都是布尔量标签值，IF 判断当 conveyor _ direction 等于 1，这也许是拟定的某个方向，即条件成立，赋值 Light 为 0；当 conveyor _ direction 等于 0，相反方向，即条件不成立，则赋值 Light 为 1，灯被点亮，报告了此时的方向。单从逻辑结果来看，相当于梯形图的梯级逻辑如图 19-3 所示。



图 19-3　等同的梯级逻辑

注意到，:= 和 [:=] 都是赋值语句，但有所不同，:= 所赋的值是保持型的数据；[:=] 所赋的值是非保持型的数据，这使得例程中预扫描和后扫描的处理不同，非保持型的赋值在预扫描和后扫描时将被复位。SFC 常常使用语句编程，它的步的后扫描就可能（如果设定）对非保持型的数据复位。上面这个语句编程，在步的后扫描执行复位操作时，会将 [:=] 所赋的值 1 复位。这正如同图 19-3 中的梯级逻辑非保持型位输出指令 OTE 也会被复位一样。图 19-2 所示的这段语句中，赋值为 1 的 Light 标签在本段语句扫描结束时会复位为 0；赋值为 0 的 Light 标签仍然为 0。所以，该段语句想要得到的结果是：一旦执行代码离开扫描，Light 的灯总是熄灭的。

语句编程的赋值语句既可以对布尔量标签赋值布尔量，也可以对数据量标签赋值数据量，这相当于梯形图逻辑代码执行的位指令 OTE 和字指令 MOV 的综合运用，并可以利用非保持型数据的复位特性，让布尔量或数据量都被清零，在梯形图中数据量后扫描清零是不能做到的。

下面让我们来看一段关于速度参数处理的语句编程实例。首先令例程首次扫描时清除速度参数 Speed _ Reference0，可以编写如图 19-4 所示的语句。

```
IF S:FS THEN
   Speed_Reference0 :=0;
END_IF;
```

图 19-4　清零的语句

这等同于如图 19-5 所示的梯级逻辑。一个简单的例程首次扫描的梯级逻辑执行，将标签 Speed _ Reference0 清除为 0。

下面的语句将根据条件对速度参数 Speed _

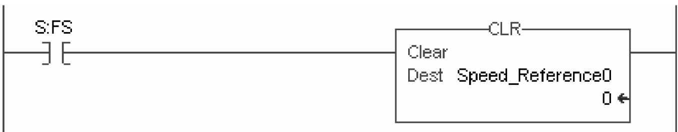


图 19-5　等同的梯级逻辑

Reference0 进行增数操作、减数操作或不改变数据的操作，如图 19-6 所示。当增数条件逻辑标签 INC _ SPD0 为 1，标签 Speed _ Reference0 加 0. 001；当减数条件逻辑标签 DEC _ SPD0 为 1，标签 Speed _ Reference0 减 0. 001；当增数和减数条件均不成立时，标签 Speed _ Reference0 直接传送，不改变数据。

```
If INC_SPD0 then
        Speed_Reference0 := Speed_Reference0+.001;
    elsif DEC_SPD0 then
        Speed_Reference0 := Speed_Reference0-.001;
    else
        Speed_Reference0 := Speed_Reference0;
End_if;
```

图 19-6　语句将根据条件对速度参数 Speed _ Reference 进行不同的操作

同样的，如果用梯级逻辑实现，如图 19-7 所示。当增数梯级条件成立，加法指令实现增数 0. 001 的操作；当减数梯级条件成立，减法指令实现减数 0. 001 的操作；当梯级条件是以上两者都不成立，则直接传送。梯形图的增数操作通常不会这样简单地提供梯级条件，因为一旦梯级条件成立，每当梯级被扫描一次，就会完成一次累加的动作。

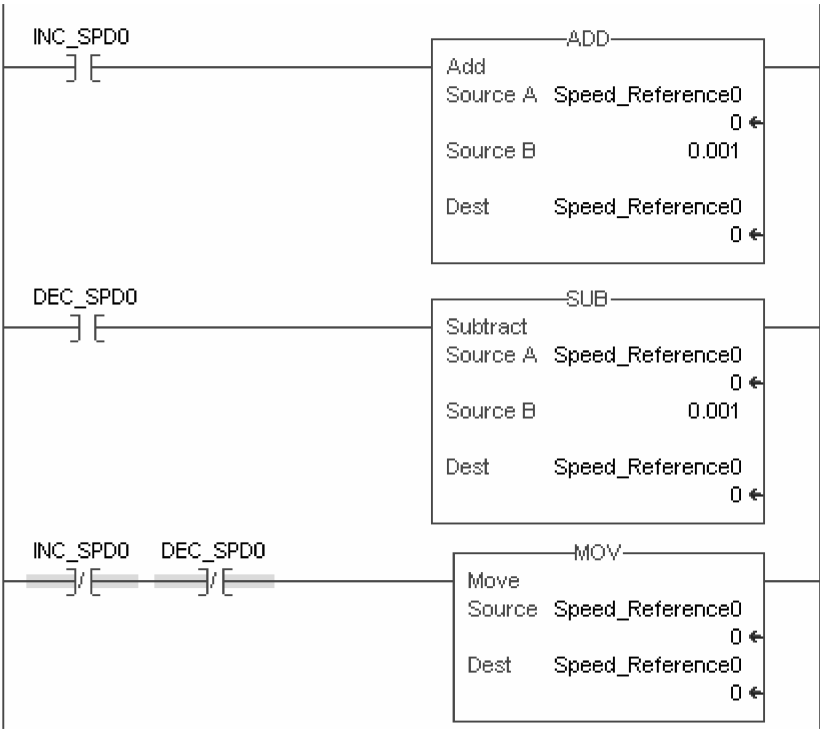


图 19-7　等同的梯级逻辑

我们再来看一段 IF THEN 语句编程实现的限幅，如图 19-8 所示。语句代码的执行将限定速度参数标签 Speed _ Reference0 的数值范围在 0 ~ 10 之内，如果 Speed _ Reference0 小于 0，则赋值为 0；如果 Speed _ Reference0 大于 10，则赋值为 10。

```
If Speed_Reference0 < 0 then
    Speed_Reference0 := 0;
End_If;

If Speed_Reference0 >= 10 then
    Speed_Reference0 := 10;
End_If;
```

图 19-8　IF THEN 语句编程实现的限幅

若将以上语句执行代码转为梯形图逻辑，则如图 19-9 所示。这正是我们在梯形图中最常见的限幅编程。

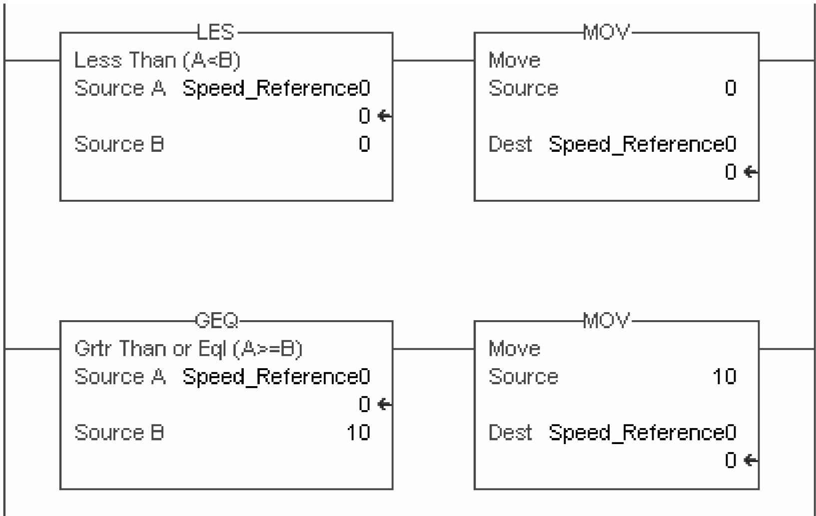


图 19-9　等同的梯形图逻辑

综合以上语句编程，实例编写如图 19-10 所示。夹杂在语句中的成对的/* 和 */，以及//是注释符号，被界定的字母是文字说明，在语句中呈绿色，这些跟 ASCII 码相似的文字是不会被扫描执行的，有时编程人员为了测试，需要屏蔽一些代码的执行，可以借助于注释符号来实现。我们知道，当一个项目文件下载到控制器中时，所有的注释文字是不会被下载到控制器中的，惟独语句编程中的注释文字是个例外，它会随着语句执行代码一起下载到控制器中，但这些注释也只能是字母文字，汉字显然不行。

语句编程最常用的是赋值语句和 IF THEN 条件判断语句，作为语句编程的一般性了解，其他的结构语句我们就不详细介绍了。

```
// Set Speed_Reference0 to Zero on First Scan of Routine

If S:FS then
Speed_Reference0 := 0;
End_IF;


        /* Speed Increase and Decrese for Speed Reference0 (Ch0) */
If INC_SPD0 then
        Speed_Reference0 /* Ch0 */:= Speed_Reference0+.001; // Increase Speed
    elsif DEC_SPD0 then
        Speed_Reference0 /* Ch0 */:= Speed_Reference0-.001; // Deacrease Speed
    else
        Speed_Reference0 /* Ch0 */:= Speed_Reference0;
End_if;

        /* Limit the Value of Speed_Reference0 between
        Zero and Ten Counts */

If Speed_Reference0 < 0 then
    Speed_Reference0 := 0;
End_If;

If Speed_Reference0 >= 10 then
    Speed_Reference0 := 10;
End_If;
```

图 19-10　语句编写的实例

19. 2　指令语句编程

语句编程的适应性强表现在很多情况下可以跟梯形图指令或功能块互换，这就是指令语句编程。我们先来看看两个熟悉的例子，了解语句编程与梯形图指令或功能块的互换关系。

关于增减计数器功能块的操作，如图 19-11 所示。

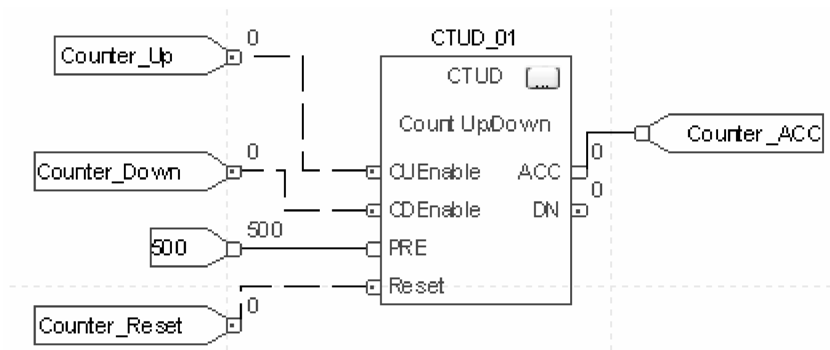


图 19-11　增减计数器功能块

若换用语句编程来完成相同的动作，如图 19-12 所示。语句执行的结果与增减计数器功

能块是一样的。

被命名为 CTUD _ 01 的计数器结构标签被赋值语句逐个赋值，有的是立即数的赋值，有的是标签的赋值，等同于功能块中的数据连接。然后，一条指令语句 CTUD（CTUD _ 01）直接指定了功能块。

```
CTUD_01.PRE := 500;
CTUD_01.Reset := Counter_Reset;
CTUD_01.CUEnable := Counter_Up;
CTUD_01.CDEnable := Counter_Down;
CTUD(CTUD_01);
Counter_ACC := CTUD_01.ACC;
```

图 19-12　语句编程完成的增减计数

还有一个比较有趣的例子是脉冲前沿执行的 OSRI 功能块和脉冲后沿执行的 OSFI 功能块，同样是前面章节的例子，如图 19-13 所示。

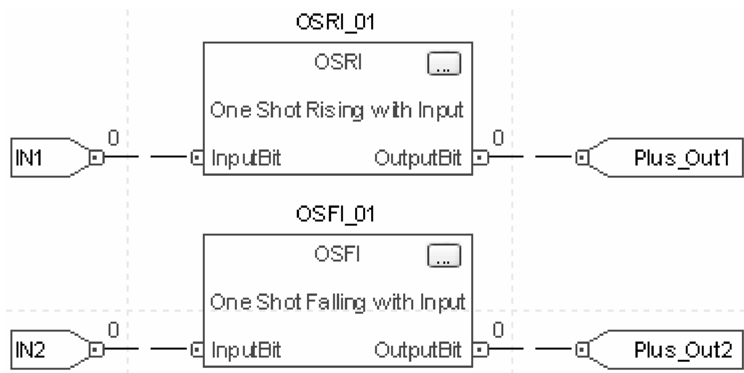


图 19-13　前沿触发和后沿触发的功能块

编写成语句，即为如图 19-14 所示。两条赋值语句分别将外部的标签与功能块的参数关联，功能块用代码 OSRI 和 OSFI 加上括号中指定的功能块结构数据标签即可。

```
OSRI_01.InputBit := IN1;
OSRI(OSRI_01);
Plus_Out1 := OSRI_01.OutputBit;

OSFI_01.InputBit := IN2;
OSFI(OSFI_01);
Plus_Out2 := OSFI_01.OutputBit;
```

图 19-14　语句编程完成的前沿触发和后沿触发

指令语句编程虽然是一条指令语句便指定了执行动作和执行对象，但是围绕这条指令的执行却需要一些相关的数据处理，所以指令语句一般是不会单独存在的。但是相关数据处理的方式是不同的，以上的两个例子是最常见的赋值语句。

无须相关数据的处理，在梯形图指令的互换中最常见的就是调用子例程指令 JSR，如图

19-15 所示的梯级逻辑。若使用语句编程实现，就是如图 19-16 所示的语句结构。这样的语句通常在 SFC 的步中出现，以实现调用一个子例程。可以看到，一条指令在此只是一条语句，括号中便是指令的参数，如果指令有多个参数，则顺延排列在圆括号中。譬如说，调用子例程如果有带入参数和带出参数，就顺延排列在例程名称之后，这正是我们在梯形图指令中所做的参数设定。

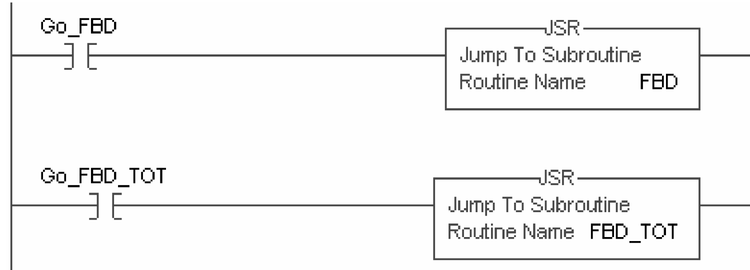
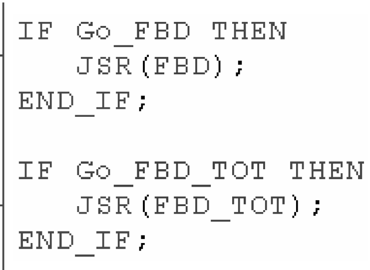


图 19-15　调用子例程指令 JSR 的梯级逻辑



```
IF Go_FBD THEN
   JSR(FBD);
END_IF;

IF Go_FBD_TOT THEN
   JSR(FBD_TOT);
END_IF;
```

图 19-16　语句编程实现的子例程调用

即使是较为复杂的指令，例如 MSG 指令的执行，同样可以由语句编程来实现，如图 19-17 所示，将令 MSG 指令反复地执行，若改为语句编程，如图 19-18 所示。看上去语句编程是非常简单的，参数只有一个，梯形图的 MSG 指令也只有 MESSAGE 结构数据标签一个参数。

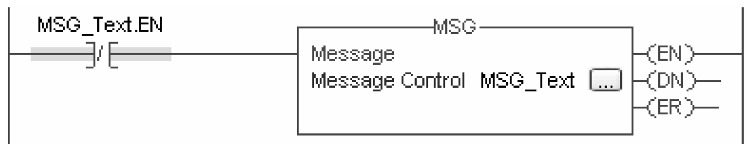
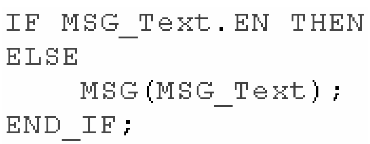


图 19-17　MSG 指令的执行由梯形图逻辑来实现



```
IF MSG_Text.EN THEN
ELSE
    MSG(MSG_Text);
END_IF;
```

图 19-18　语句编程实现的 MSG 执行

但是，MSG 指令除了编写执行的逻辑外，还需要对指令的执行内容进行组态，不但要组态访问对象的数据标签，还要组态通信路径和连接方式，这些都需要通过组态页面来实施。语句编程方式又是如何进入到组态的页面呢？这在离线文件中是不能操作的，只有将项目下载到控制器，处于在线监视状态才有可能。如图 19-19 所示，连接在线的控制器项目，在语句编程部分选中 MSG 指令执行语句的结构数据标签 MSG _ Text，点击右键，选择 Configure“MSG _ Text”，点击便可进入 MSG 指令的组态页面了，可方便地在线进行监视和修改。

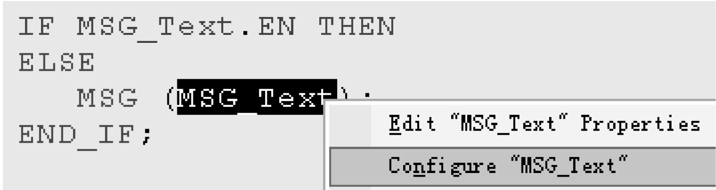


图 19-19　在线对 MSG 组态

在梯形图逻辑编程的 MSG 指令不管是在线还是离线，都有一个组态页面提供给我们去

完成所有的有关通信的组态。语句编程如果想在离线时进行通信组态，不得不到数据库的 MESSAGE 结构标签中去设置，这的确不够直观，只有对 MSG 指令和 MESSAGE 结构数据标签相当熟悉和了解才能找到相应的参数。在这种情况下，我建议借助于梯形图指令对这个 MSG 的结构数据标签完成组态，不管是梯形逻辑执行的 MSG 还是语句执行代码执行的都是一样。

举一反三，指令的语句编程，只要是在梯形图中具有组态页面的，都可以在线监视和修改，请尝试采用上述 MSG 指令组态的方式来操作。

语句编程还有一点值得注意的是，只要被扫描到，满足条件的语句就会被执行，对于梯形图中需要触发才能执行的指令，在结构语句中很可能每次扫描都会执行，要确保惟一性操作，就要特别地编写 OSRI 或 OSFI 指令来给予限制。这也是语句编程的不同之处。

语句编程最大的特点就是适应性强，可以编写出很多功能代码，代替指令或功能块的实现，执行速度也足够快，可惜直观性不够，尤其是遇到大量数据需要组态时，离线无法提供一个直观的组态页面。从这一点来看，指令和功能块的优势便更为明显，MSG 指令就是一个很说明问题的例子。在 PAC 控制器中组合梯形图、功能块和语句编程的特长来运用，是 Logix 控制器的产品优势之一。

当用语句编程互换梯形图指令或者功能块时，是用梯形图指令互换，还是用功能块互换，应该查看在线指令帮助文件，那里列举了互换的关系。尽管我们说语句编程就是将梯形图指令或功能块的代码加上括号内的参数组成，但实际上究竟是互换梯形图指令还是互换功能块是不能主观臆测的，例如宽脉冲转为窄脉冲的单脉冲触发，语句编程是只能跟功能块互换而不能跟梯形图指令互换的。

第 20 章

报警功能的三种编程模式

至此，我们已经学习了三种执行代码的编程模式。对于相同或相近的功能动作，有的可以用梯形图指令和功能块实现；有的可以用梯形图指令和语句实现；有的可以用功能块和语句实现；还有的用梯形图指令、功能块和语句都可以实现。

较高版本的控制器所增加的报警功能操作就是用三种执行代码都可以编程实现的。这里，我们不妨以此为例，对比一下实现同一功能动作的三种编程模式，也算是对 Logix 控制器编程语言的一个总结。

实际需求中，有现场监测状态引起的报警，称为离散量报警；也有模拟量超出监测上限和监测下限引发的报警，称为模拟量报警。报警发生后往往需要确认或复位，报警触发延时一段时间以后才产生报警动作。报警功能的执行代码便是针对此需求而开发的。

20.1 离散量报警功能的编程

离散量报警源于现场开关状态，当被监视的开关状态变化，作为报警触发动作，可以是前沿触发，也可以是后沿触发，一般在延时一段时间之后，并非干扰信号，确认为真实的报警信号，然后作出报警的处理。

编写的离散量报警功能的梯级逻辑如图 20-1 所示。梯级条件 Alarm _ Digital 就是报警触发条件，在外部编程梯级逻辑，用以控制离散量报警功能。

报警指令的程控操作数：

- ALMD 的报警确认 AlarmD _ Ack 为 1 执行报警确认；
- 报警复位 AlarmD _ Reset 为 1 执行报警复位；
- 报警取消 AlarmD _ Disable 为 1 取消 ALMD 指令报警功能；
- 报警使能 AlarmD _ Enable 为 1 使能 ALMD 指令报警功能。

报警指令的输出状态：

- 报警状态 InAlarm 为 1 表示处在报警中，为 0 表示未报警；
- 报警已确认 Acked 为 1 表示报警已被确认，为 0 表示未被确认；
- 报警禁止 Suppressed 为 1 表示报警被禁止，为 0 表示未被禁止；
- 报警消除 Disabled 为 1 表示报警取消，为 0 表示报警使能；
- 报警指令故障 InstructFault 为 1 表示指令故障，为 0 表示指令正常。

报警指令运行过程所显示的参数：

- MinDurationPRE 最小持续时间的预置值；
- MinDurationACC 最小持续时间的累加值。

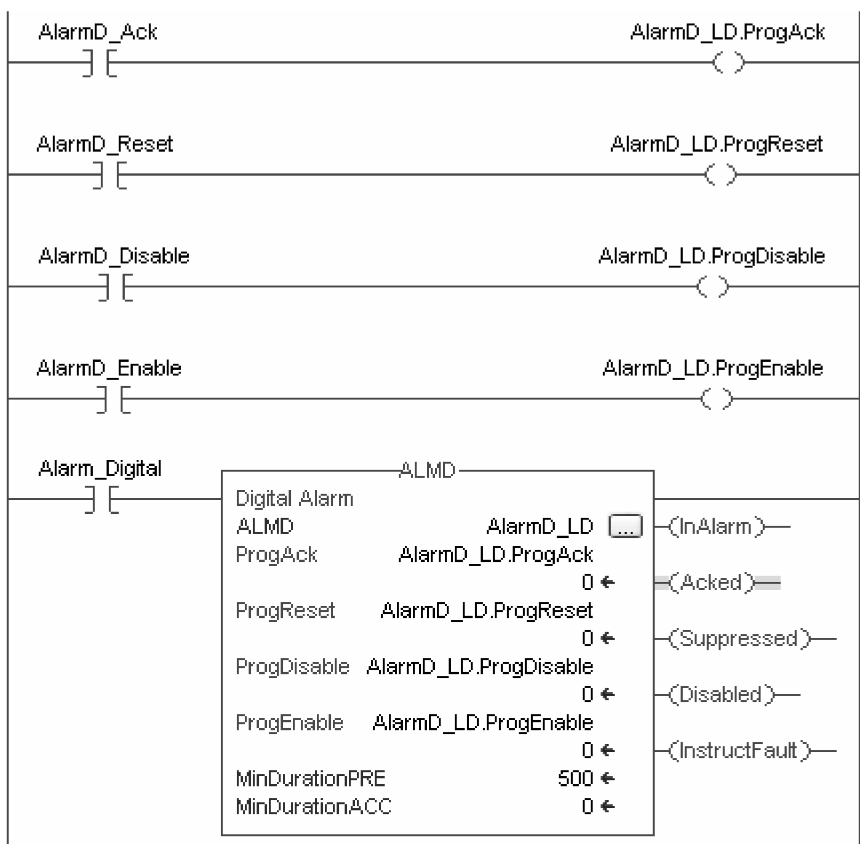


图 20-1　离散量报警功能的梯级逻辑

点击指令右上角的[...]，展开 ALMD 指令的组态页面，如图 20-2 所示。组态页面也是离散量报警结构数据标签 AlarmD _ LD 的数据界面。在组态页面组态的信息将存放在结构数据标签 AlarmD _ LD 中，参数页面的信息也将存放在结构数据标签 AlarmD _ LD 中。在创建标签的数据库中，我们能看到大量的参数，在指令面板上出现的只是显示参数。

组态页面参数的说明如下：

- Condition（报警触发条件）：布尔量数据，可设置为 Input = 1，则当输入条件为 1 报警触发；或设置为 Input = 0，则当输入条件为 0 报警触发。
- Severity（报警等级范围）：可选则 1 ~ 1000，作为报警等级，当多个报警同时触发时的优先级别设定。
  - 设定 1 ~ 250 为低优先级别；
  - 设定 251 ~ 500 为中优先级别；
  - 设定 501 ~ 750 为高优先级别；
  - 设定 751 ~ 1000 为紧急优先级别。

此处设置 650 为报警级别，即高优先级别。

- Minimum Duration（最小持续时间）：报警条件触发至少要持续这段时间才产生报警报告信息。此处设定为 500ms，即报警条件持续了 500ms 之后，才会产生报警动作。

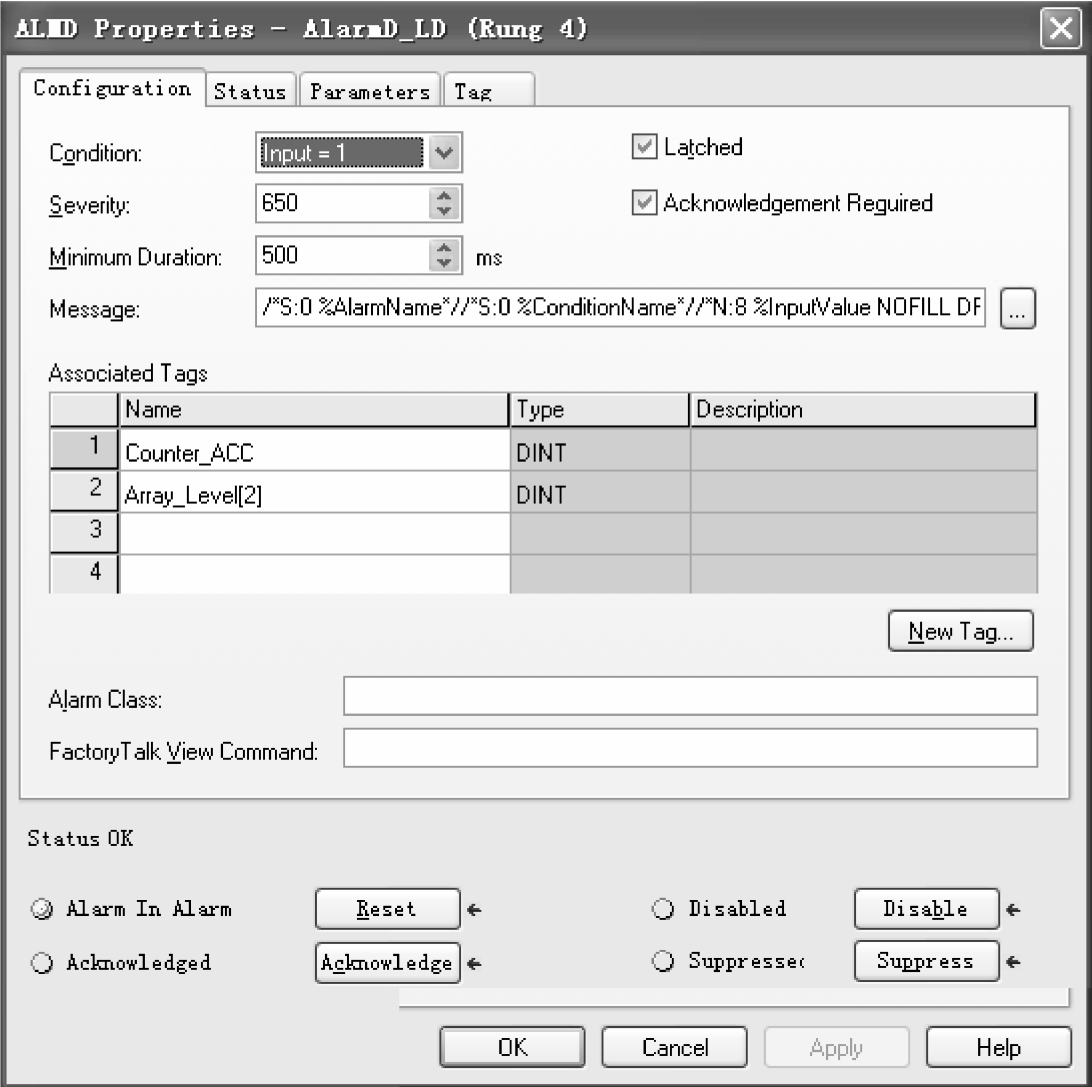


图 20-2　ALMD 指令的组态页面

- Latched（锁定报警）：勾选此项则报警发生时要锁定，即使报警条件已经消失，也仍然维持报警，直到接收到复位命令；如果报警条件未消失，则复位命令不起作用。
- Acknowledgement Required（确认需求）：勾选此项，该报警需要被确认，当报警发生后，不仅报警触发条件消失，还要报警确认后才能消除报警状态。
- Message（报警信息）：为报警信息的显示内容或显示形式组态，点击[...]，进入信息编辑的组态页面，如图 20-3 所示。可以在 Add Variable 中挑选要显示的选项，如报警名称、条件名称、输入量、限量、安全优先等级等。如图 20-3 中右侧显示的是安全等级显示信息形式的组态，将用 5 位数来表达安全等级，左边以空格充填，编辑完毕点击 Add 便可加入显示栏。也可以在 Message 栏目中直接键入要显示的报警内容文字，即在/*……*/之间键入要表达的信息。信息栏目所显示的信息总量

不能超过 255 个字符。

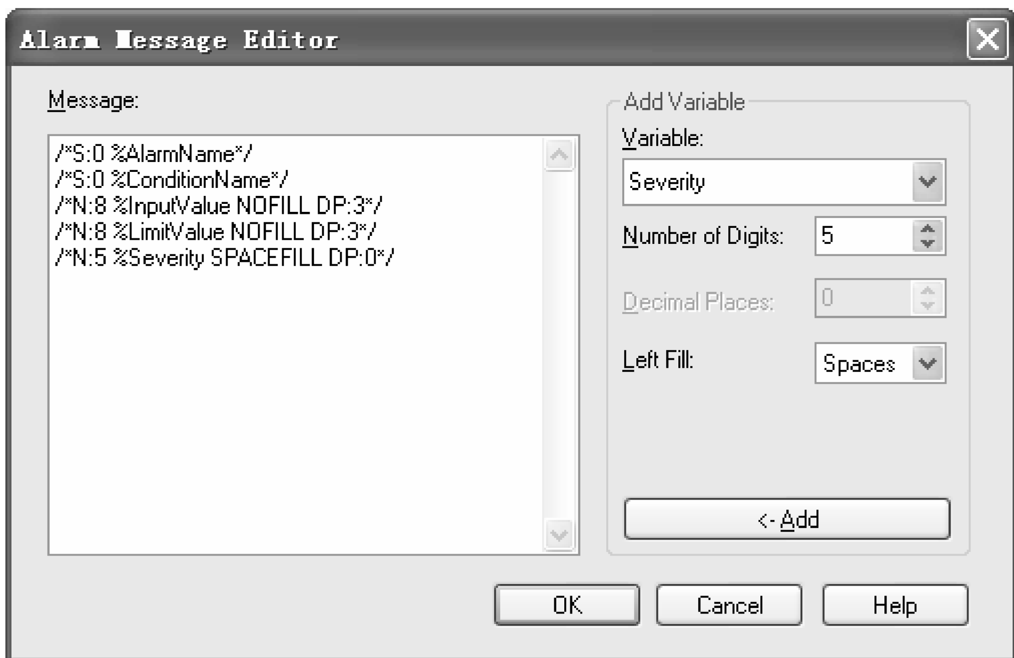


图 20-3　信息编辑的组态页面

- Associated Tags（关联标签）：指定最多 4 个与报警有关联的标签，任何时候，这些标签被存储的值映射于此，报警条件变成报警状态时，或其他报警状态发生，这些存储值被传送给报警。这些存储的值也可以被内含在报警信息中。关联标签可以是基本数据或字符串，也可以是用户自定义标签或数组的子元素。此例中是一个基本数据双整字标签 Counter _ ACC 和一个数组标签元素 Array _ Level ［2］。
- Alarm Class（报警分类）：用报警分类来组合相关的报警，例如 Tank Farm A 组合了油罐的所有报警信息；Control Loop 组合了所有的 PID 环的报警。报警分类命名必须精确符合，并区分大小写。在人机界面，可以通过报警分类来过滤，例如操作员只显示类别是 Tank Farm A 和 Control Loop 的所有信息。
- FactoryTalk View Command：键入一个 FactoryTalk View 命令，这个命令在操作员站执行，当操作员看到指定报警时操作。ALMD 指令为 FactoryTalk View 提供了附加功能，在某个报警发生并通过面板显示给操作员，操作员根据提示按下按钮，运行一个相关的命令。命令执行通常是弹出特定的面板和显示、执行一个宏、访问帮助文件或启动一个外部应用。命令键入没有错误校验，请务必仔细，错误的命令将不会执行。

在报警锁定和报警确认需求选项都被勾选时发生报警，各个报警状态位的波形图如图 20-4 所示。每当报警条件成立，延时一小段时间才会给出报警状态。只有在报警已确认且报警条件消失时，报警复位才能消除报警状态。

如图 20-5 所示是状态页面。该页面显示了最后一次报警经历的时间记录以及报警次数记录，累积次数可复位。

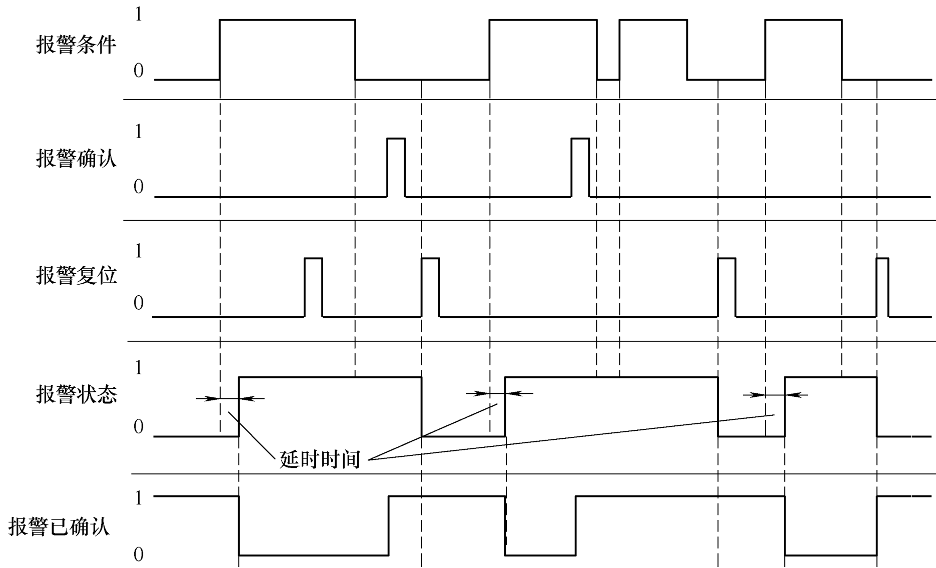


图 20-4　各个报警状态位的波形图

如图 20-6 所示是参数页面的部分输入参数。在此组态页面所见到的只是其中很少的一部分，关于每个参数的含义和运用，在罗克韦尔出版物的指令集中可以找到详细的介绍。

如图 20-7 所示是参数页面的部分输出参数。在此组态页面所见到的只是其中很少的一部分，关于每个参数的含义和运用，在罗克韦尔出版物的指令集中可以找到详细的介绍。

如图 20-8 所示是报警条件 Alarm _ Digital 成立，ALMD 指令执行，延时 500ms 之后的状态。可以看到 Acked 被复位，InAlarm 被置位，报警状态送出。即使报警条件已经不成立，未经确认和复位，报警状态是不会消失的。

离散量报警的功能块组态如图 20-9 所示。跟梯形图指令是一样的，其操作数也可被程控。

外部显示参数：

- 报警条件 Alarm _ Digital 为 BOOL 量，接入报警 In；
- 报警确认程控操作数 AlarmD _ Ack，接入 ProgAck；
- 报警复位程控操作数 AlarmD _ Reset，接入 ProgReset；
- 报警取消程控操作数 AlarmD _ Disable，接入 ProgDisable；
- 报警使能程控操作数 AlarmD _ Enable，接入 ProgEnable；
- 报警状态 InAlarm 为 1 表示处在报警中，为 0 表示未报警；
- 报警已确认 Acked 为 1 表示报警已被确认，为 0 表示未被确认；
- 报警禁止 Suppressed 为 1 表示报警被禁止，为 0 表示未被禁止；
- 报警取消 Disabled 为 1 表示报警取消，为 0 表示报警使能。

点击功能块右上角的[...]展开组态页面，跟梯形图指令的组态页面完全一样，如图 20-2 所示。状态页面如图 20-5 所示。组态参数含义也完全相同，参数的页面也跟梯形图指令的

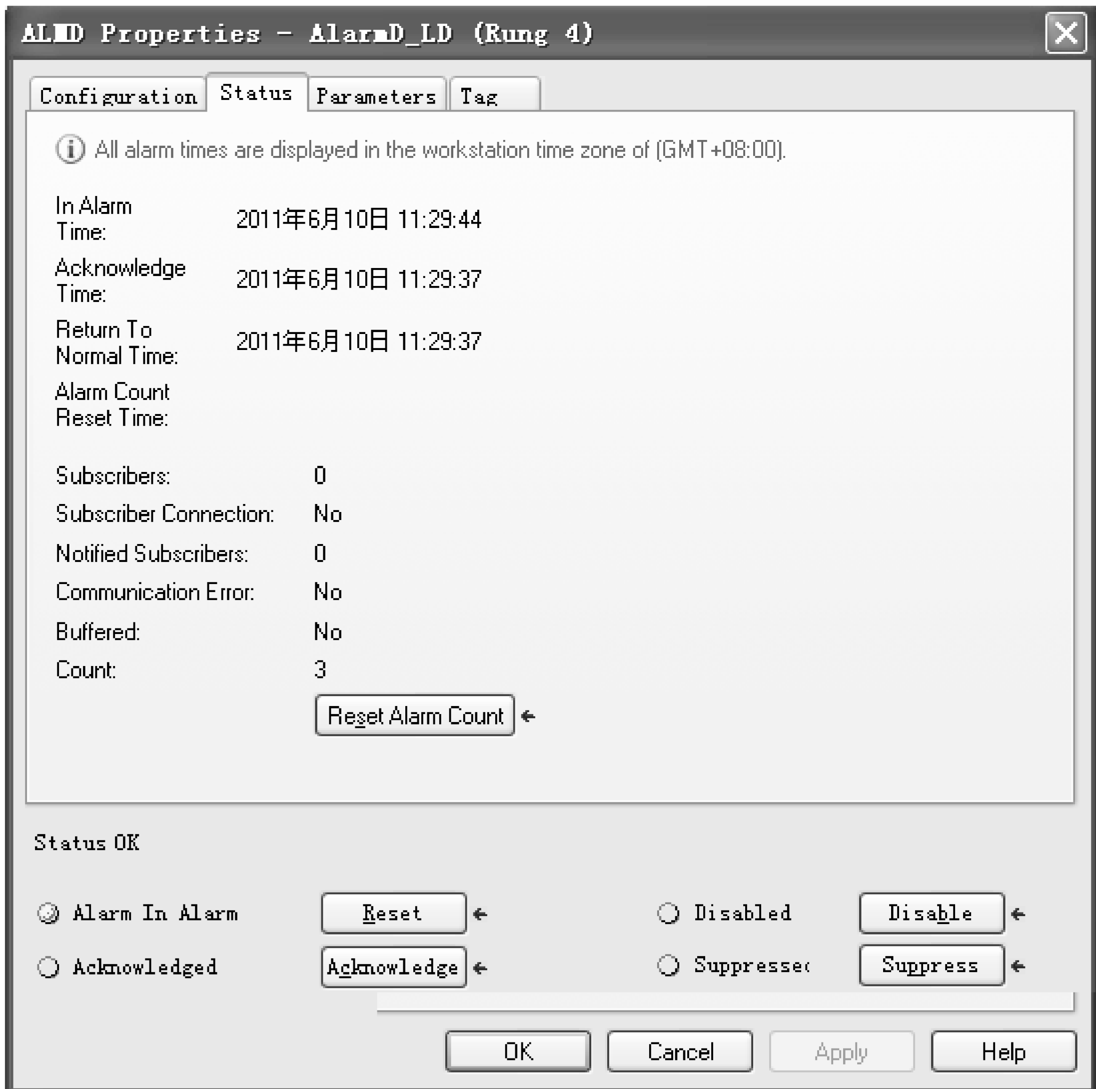


图 20-5　离散量报警状态页面

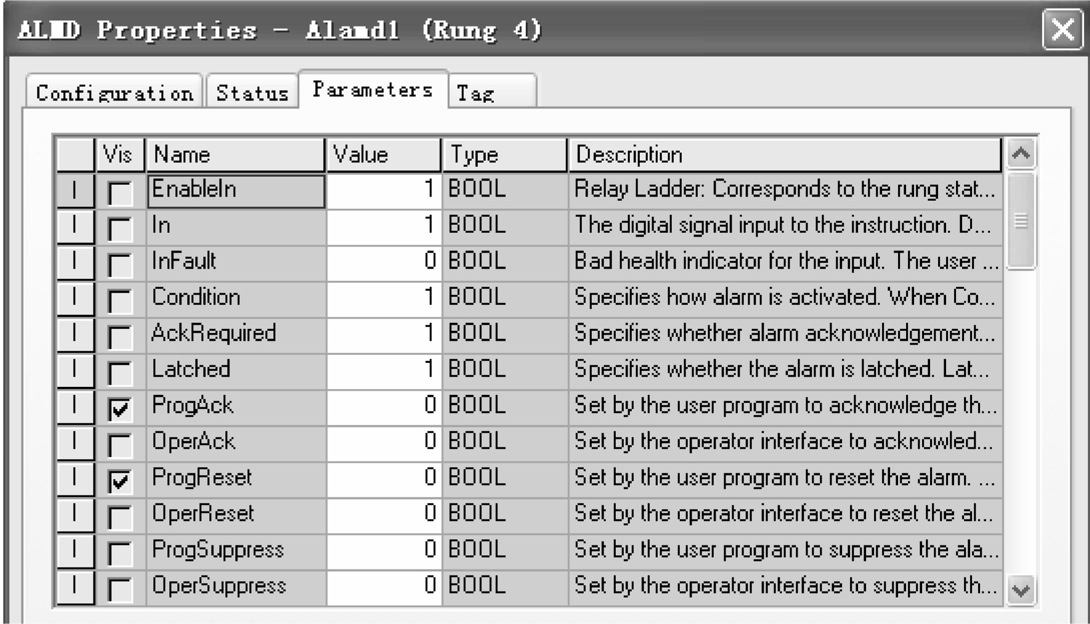
完全一样，如图 20-6 和图 20-7 所示。这里不再赘述。

功能块 ALMD 执行的结果如图 20-10 所示。当报警输入 Alarm _ Digital 为 1 时，经过 500ms 的延时，功能块输出报警状态，InAlarm 输出为 1，且 Acked 为 0。

也许是梯形图指令编程模仿了功能块的编程和组态，我们看到它们是如此的相似，这是报警各项功能齐全，并且高度集中的专业性指令，而且报警指令与人机界面有特定的关联，报警数据结构内嵌在上位机报警组态的内容中。

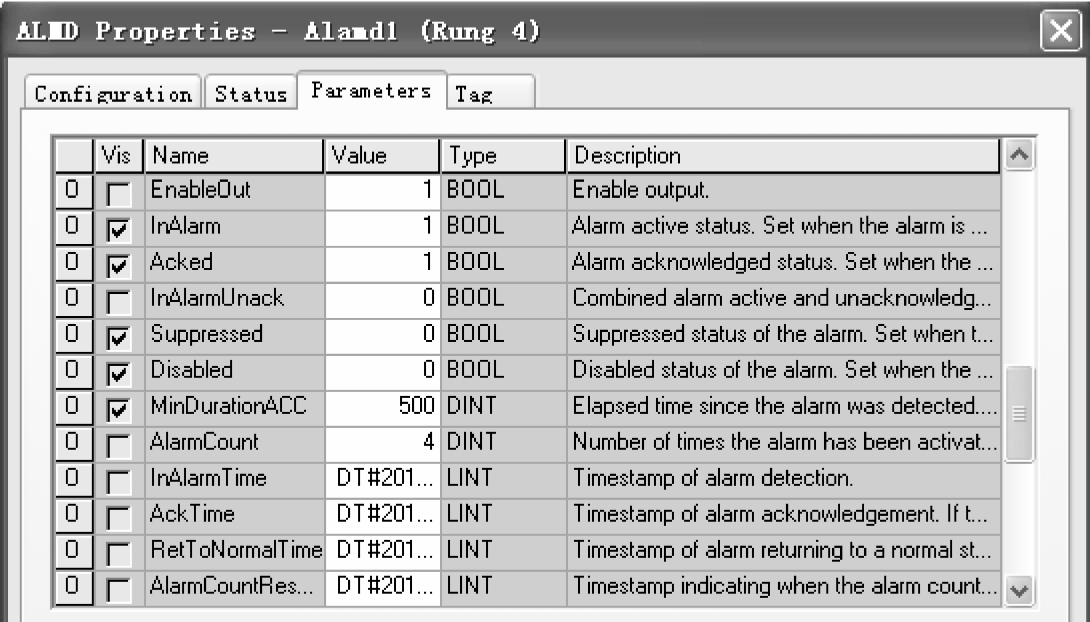
最后，我们再来看看报警功能的语句编程。首先，在数据库创建一个离散量报警的结构数据标签，命名为 AlarmD _ Text，如图 20-11 所示。

编写语句报警逻辑如图 20-12 所示。赋值语句完成了离散量报警的报警确认、报警复位、报警使能和报警取消的程控操作，指令 ALMD 及括号中相关的参数顺序而列。



	Vis	Name	Value	Type	Description
I	☐	EnableIn	1	BOOL	Relay Ladder: Corresponds to the rung stat...
I	☐	In	1	BOOL	The digital signal input to the instruction. D...
I	☐	InFault	0	BOOL	Bad health indicator for the input. The user ...
I	☐	Condition	1	BOOL	Specifies how alarm is activated. When Co...
I	☐	AckRequired	1	BOOL	Specifies whether alarm acknowledgement...
I	☐	Latched	1	BOOL	Specifies whether the alarm is latched. Lat...
I	☑	ProgAck	0	BOOL	Set by the user program to acknowledge th...
I	☐	OperAck	0	BOOL	Set by the operator interface to acknowled...
I	☑	ProgReset	0	BOOL	Set by the user program to reset the alarm. ...
I	☐	OperReset	0	BOOL	Set by the operator interface to reset the al...
I	☐	ProgSuppress	0	BOOL	Set by the user program to suppress the ala...
I	☐	OperSuppress	0	BOOL	Set by the operator interface to suppress th...

图 20-6　参数页面的部分输入参数



	Vis	Name	Value	Type	Description
O	☐	EnableOut	1	BOOL	Enable output.
O	☑	InAlarm	1	BOOL	Alarm active status. Set when the alarm is ...
O	☑	Acked	1	BOOL	Alarm acknowledged status. Set when the ...
O	☐	InAlarmUnack	0	BOOL	Combined alarm active and unacknowledg...
O	☑	Suppressed	0	BOOL	Suppressed status of the alarm. Set when t...
O	☑	Disabled	0	BOOL	Disabled status of the alarm. Set when the ...
O	☑	MinDurationACC	500	DINT	Elapsed time since the alarm was detected....
O	☐	AlarmCount	4	DINT	Number of times the alarm has been activat...
O	☐	InAlarmTime	DT#201...	LINT	Timestamp of alarm detection.
O	☐	AckTime	DT#201...	LINT	Timestamp of alarm acknowledgement. If t...
O	☐	RetToNormalTime	DT#201...	LINT	Timestamp of alarm returning to a normal st...
O	☐	AlarmCountRes...	DT#201...	LINT	Timestamp indicating when the alarm count...

图 20-7　参数页面的部分输出参数

语句编程的报警页面组态不能在离线情况下完成，只有在线控制器中可直接进行组态。在语句指令圆括号的参数项中，选中报警结构标签 AlarmD _ Text，点击右键，选择 Configure “AlarmD _ Text”，点击进入，即可进入跟梯形图指令和功能块完全一样的组态界面，从而完

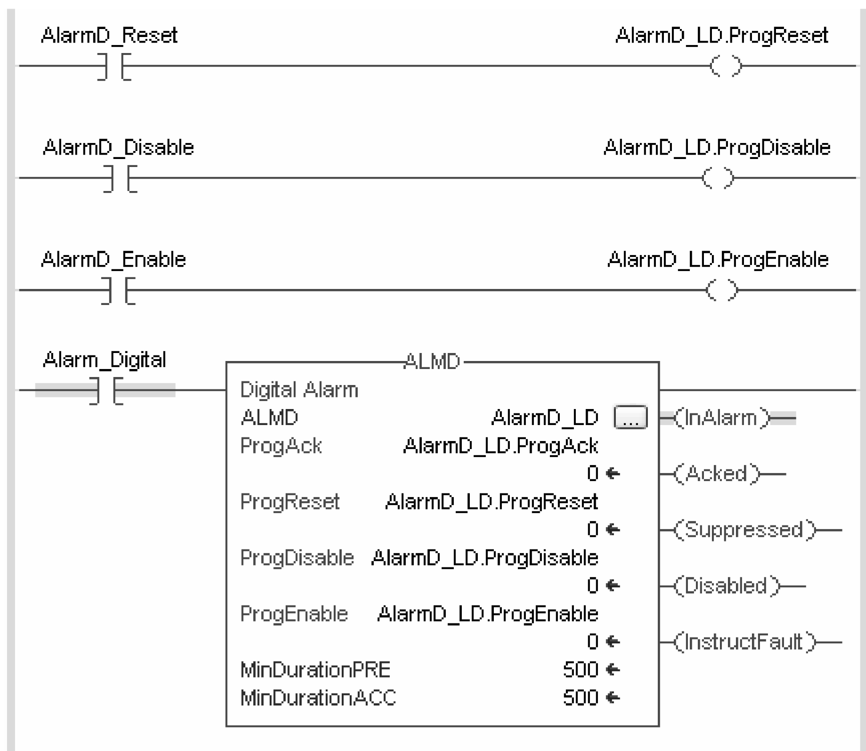


图 20-8　离散量报警梯级逻辑

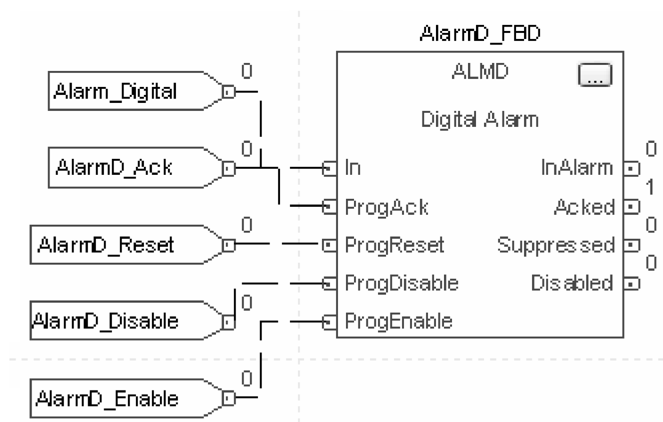


图 20-9　离散量报警的功能块组态

成组态工作。语句编程的报警页面组态的进入如图 20-13 所示。

显然，不管是梯形图指令编程、功能块编程，抑或是语句编程，离散量报警结构数据都是一样的，不但有特定的组态界面，可直观地完成，也可以在报警结构数据中设置。位于数据区域的离散量报警结构数据标签如图 20-14 所示。如果熟练的话，直接在结构数据表中设置有关参数也未尝不可。

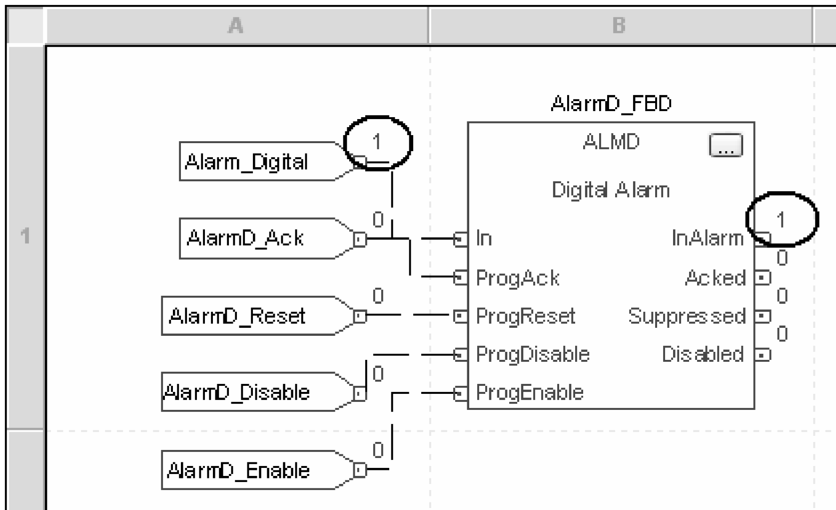


图 20-10　功能块 ALMD 执行的结果

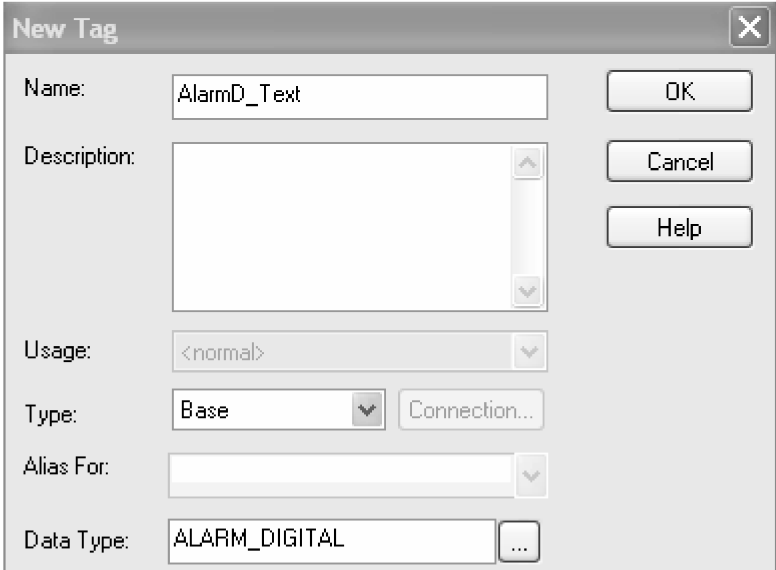


图 20-11　在数据库创建一个离散量报警的结构数据标签

```
AlarmD_Text.In := Alarm_Digital;
AlarmD_Text.ProgAck := AlarmD_Ack;
AlarmD_Text.ProgReset := AlarmD_Reset;
AlarmD_Text.ProgDisable := AlarmD_Disable;
AlarmD_Text.ProgEnable := AlarmD_Enable;
ALMD (AlarmD_Text,AlarmD_Text.In,AlarmD_Text.ProgAck,
AlarmD_Text.ProgReset,AlarmD_Text.ProgDisable,
AlarmD_Text.ProgEnable);
```

图 20-12　编写语句离散量报警

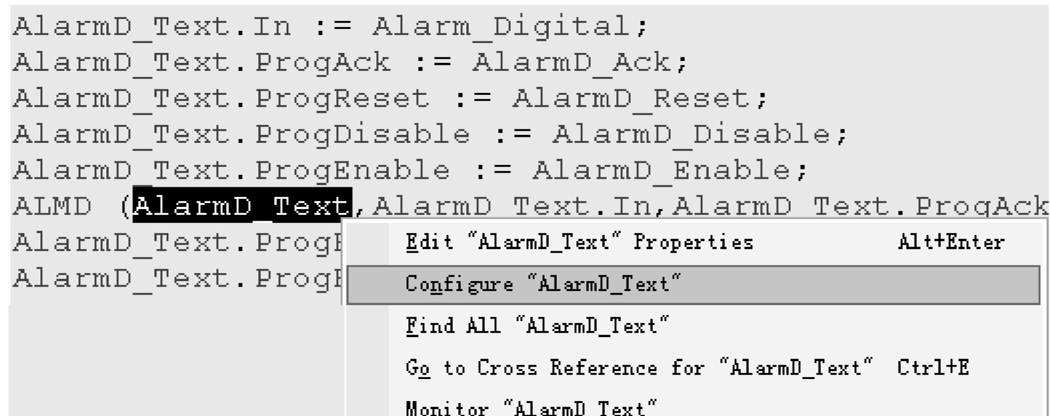


图 20-13　语句编程的报警页面组态的进入

Name	Value	F	Style	Data Type
AlarmD_Text	{...}	{.		ALARM_DIGI...
AlarmD_Text.EnableIn	0		Decimal	BOOL
AlarmD_Text.In	0		Decimal	BOOL
AlarmD_Text.InFault	0		Decimal	BOOL
AlarmD_Text.Condition	1		Decimal	BOOL
AlarmD_Text.AckRequired	1		Decimal	BOOL
AlarmD_Text.Latched	1		Decimal	BOOL
AlarmD_Text.ProgAck	0		Decimal	BOOL
AlarmD_Text.OperAck	0		Decimal	BOOL
AlarmD_Text.ProgReset	0		Decimal	BOOL
AlarmD_Text.OperReset	0		Decimal	BOOL
AlarmD_Text.ProgSuppress	0		Decimal	BOOL
AlarmD_Text.OperSuppress	0		Decimal	BOOL
AlarmD_Text.ProgUnsuppress	0		Decimal	BOOL
AlarmD_Text.OperUnsuppress	0		Decimal	BOOL
AlarmD_Text.ProgDisable	0		Decimal	BOOL
AlarmD_Text.OperDisable	0		Decimal	BOOL
AlarmD_Text.ProgEnable	0		Decimal	BOOL
AlarmD_Text.OperEnable	0		Decimal	BOOL
AlarmD_Text.AlarmCountReset	0		Decimal	BOOL
AlarmD_Text.UseProgTime	0		Decimal	BOOL
AlarmD_Text.ProgTime	DT#1970-...		Date/Time	LINT
AlarmD_Text.Severity	650		Decimal	DINT
AlarmD_Text.MinDurationPRE	500		Decimal	DINT

图 20-14　离散量报警结构数据标签

AlarmD_Text.EnableOut	0	Decimal	BOOL
AlarmD_Text.InAlarm	0	Decimal	BOOL
AlarmD_Text.Acked	1	Decimal	BOOL
AlarmD_Text.InAlarmUnack	0	Decimal	BOOL
AlarmD_Text.Suppressed	0	Decimal	BOOL
AlarmD_Text.Disabled	0	Decimal	BOOL
+ AlarmD_Text.MinDurationACC	0	Decimal	DINT
+ AlarmD_Text.AlarmCount	0	Decimal	DINT
AlarmD_Text.InAlarmTime	DT#1970-...	Date/Time	LINT
AlarmD_Text.AckTime	DT#1970-...	Date/Time	LINT
AlarmD_Text.RetToNormalTi...	DT#1970-...	Date/Time	LINT
AlarmD_Text.AlarmCountRes...	DT#1970-...	Date/Time	LINT
AlarmD_Text.DeliveryER	0	Decimal	BOOL
AlarmD_Text.DeliveryDN	0	Decimal	BOOL
AlarmD_Text.DeliveryEN	0	Decimal	BOOL
AlarmD_Text.NoSubscriber	0	Decimal	BOOL
AlarmD_Text.NoConnection	0	Decimal	BOOL
AlarmD_Text.CommError	0	Decimal	BOOL
AlarmD_Text.AlarmBuffered	0	Decimal	BOOL
+ AlarmD_Text.Subscribers	0	Decimal	DINT
+ AlarmD_Text.SubscNotified	0	Decimal	DINT
+ AlarmD_Text.Status	0	Decimal	DINT
AlarmD_Text.InstructFault	0	Decimal	BOOL
AlarmD_Text.InFaulted	0	Decimal	BOOL
AlarmD_Text.SeverityInv	0	Decimal	BOOL

图 20-14　离散量报警结构数据标签（续）

20.2　模拟量报警功能的编程

模拟量报警源于对现场一个数据量的监视，当被监视的数据量超越了预先设定的上限值或者下限值，触发相应的报警状态，有时是高报警状态，有时是低报警状态，一般在状态维持一段时间之后，确认为真实的报警状态，然后作出报警的处理。

模拟量报警功能的梯形图编程如图 20-15 所示。

报警指令的程控参数：

- ALMD 的报警确认 AlarmA _ AckAll 执行报警确认；
- 报警取消 AlarmA _ Disable 为 1 取消 ALMA 指令报警功能；
- 报警使能 AlarmA _ Enable 为 1 使能 ALMA 指令报警功能。

报警指令的输出状态：

- 报警状态 HHInAlarm 为 1 表示处在高高报警中，为 0 表示未报警；
- 报警状态 HInAlarm 为 1 表示处在高报警中，为 0 表示未报警；
- 报警状态 LInAlarm 为 1 表示处在低报警中，为 0 表示未报警；
- 报警状态 LLInAlarm 为 1 表示处在低低报警中，为 0 表示未报警；
- 报警状态 ROCPosInAlarm 为 1 表示处在正向变化率报警中，为 0 表示未报警；
- 报警状态 ROCNegInAlarm 为 1 表示处在反向变化率报警中，为 0 表示未报警；
- 报警已确认 HHAcked 为 1 表示高高报警已被确认，为 0 表示未确认；
- 报警已确认 HAcked 为 1 表示高报警已被确认，为 0 表示未确认；
- 报警已确认 LAcked 为 1 表示低报警已被确认，为 0 表示未确认；
- 报警已确认 LLAcked 为 1 表示低低报警已被确认，为 0 表示未确认；
- 报警已确认 ROCPosAcked 为 1 表示正向变化率报警已被确认，为 0 表示未确认；
- 报警已确认 ROCNegAcked 为 1 表示反向变化率报警已被确认，为 0 表示未确认；
- 报警禁止 Suppressed 为 1 表示报警被禁止，为 0 表示未被禁止；
- 报警消除 Disabled 为 1 表示报警取消，为 0 表示报警使能；
- 报警指令故障 InstructFault 为 1 表示指令故障，为 0 表示指令正常。

指令运行显示的组态参数：

- HHLimit：高高报警限值；
- HLimit：高报警限值；
- LLimit：低报警限值；
- LLLimit：低低报警限值。

点击指令右上角的[...]，展开 ALMA 指令的组态页面，也是模拟量报警结构数据标签 AlarmA _ LD 的数据界面，在组态页面组态的信息将存放在结构数据标签 AlarmA _ LD 中，参数页面的信息也将存放在结构数据标签 AlarmA _ LD 中。在创建标签的数据库中，我们能看到大量的参数，在指令面板上出现的只是显示参数。

如图 20-16 所示是组态页面左侧部分，是有关报警界限和延时时间的设定。

参数说明：

- High High：高高报警限值设定，勾选并键入设定值。
- High：高报警限值设定，勾选并键入设定值。
- Low：低报警限值设定，勾选并键入设定值。
- Low Low：低低报警限值设定，勾选并键入设定值。
- Severity：报警等级范围，每个限值后面自行独立设定。可选 1 ~ 1000 作为报警等级，当多个报警同时触发时的优先级别设定如下：
  - 设定 1 ~ 250 为低优先级别；
  - 设定 251 ~ 500 为中优先级别；
  - 设定 501 ~ 750 为高优先级别；
  - 设定 751 ~ 1000 为紧急优先级别。

此处设置默认值 500 为报警级别，即中优先级别。

- Minimum Duration（最小持续时间）：当满足报警条件后至少要持续这段时间才产生报警报告信息。此处设定为 3000ms，即报警条件持续了 3s 之后才产生相应的报

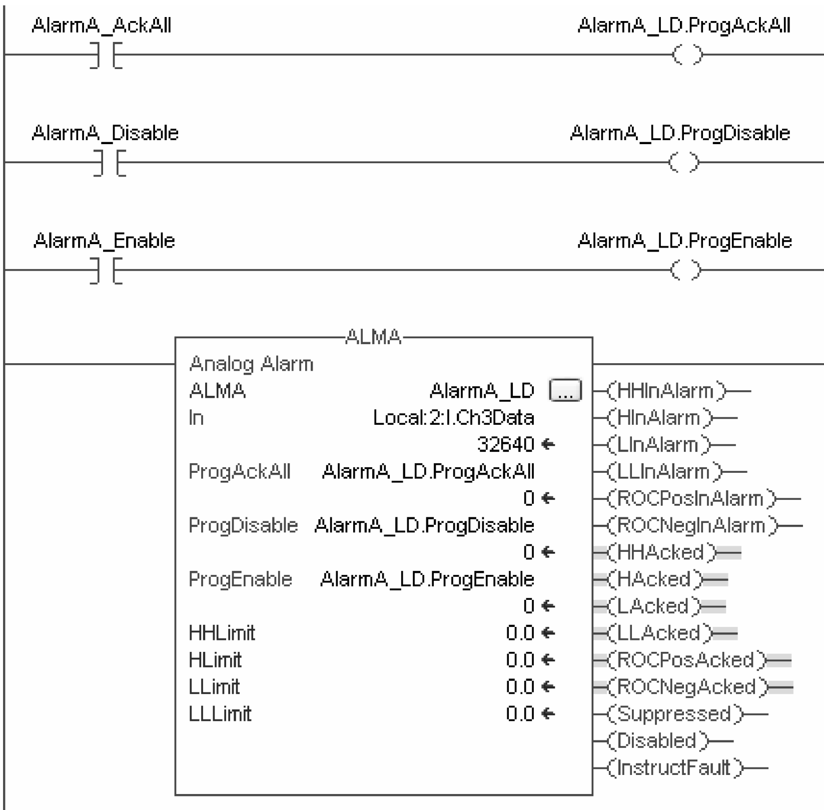


图 20-15　模拟量报警功能的梯级逻辑

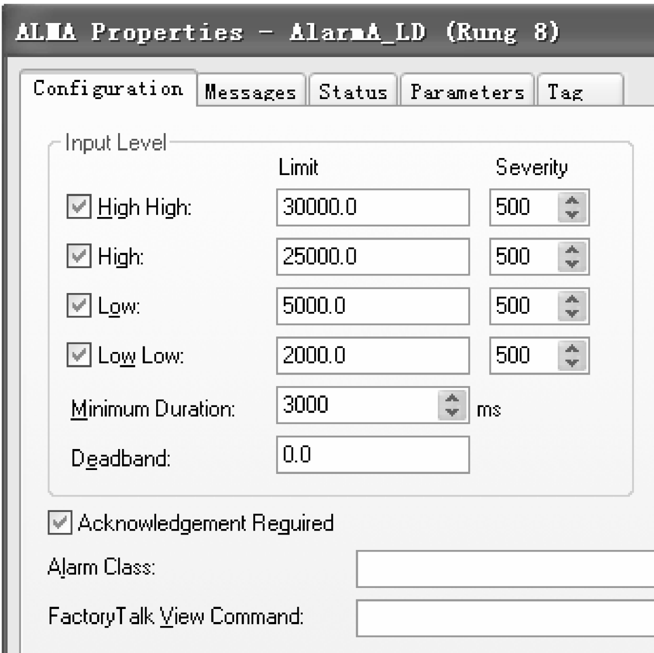


图 20-16　ALMA 指令组态页面左侧部分

警动作。

- Deadband（死区带宽）：设定死区范围。
- Acknowledgement Required（确认需求）：勾选该项，有关该报警的所有的条件都需要确认。
- Alarm Class（报警分类）：用报警分类来组合相关的报警，例如 Tank Farm A 组合了油罐的所有报警信息；Control Loop 组合了所有的 PID 环的报警。报警分类命名必须精确符合，并区分大小写。在人机界面，可以通过报警分类来过滤，例如操作员只显示类别是 Tank Farm A 和 Control Loop 的所有信息。
- FactoryTalk View Command：键入一个 FactoryTalk View 命令，这个命令在操作员站执行，当操作员看到指定报警时操作。ALMA 指令为 FactoryTalk View 提供了附加功能，在某个报警发生并通过面板显示给操作员，操作员根据提示按下按钮，运行一个相关的命令。命令执行通常是弹出特定的面板和显示、执行一个宏、访问帮助文件或启动一个外部应用。命令键入没有错误校验，请务必仔细，错误的命令将不会执行。

如图 20-17 所示是组态页面右侧部分，是有关输入信号变化率监视的设定。

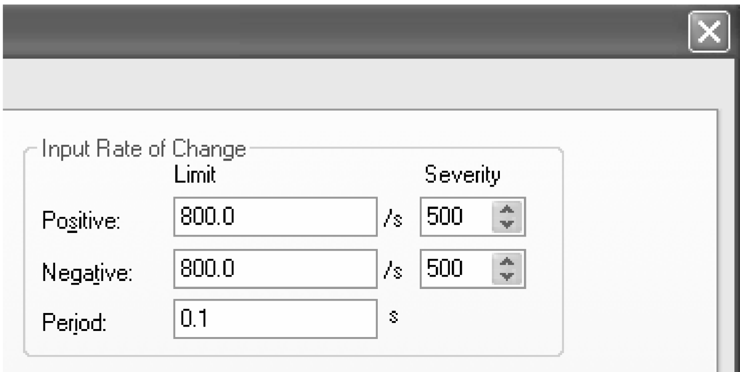


图 20-17　ALMA 指令组态页面右侧部分

参数说明：

- Postive（正向变化率报警值设定）：此处设定每秒变化 800 个单位值；
- Negative（反向变化率报警值设定）：此处设定每秒变化 800 个单位值；
- Period（变化率监测采样时间）：此处设定 100ms 的采样时间。

点击信息组态页面如图 20-18 所示，是有关信息显示的设定。

参数说明：

- Level 关于上下限值的信息说明：
  - High High：高高报警信息说明；
  - High：高报警信息说明；
  - Low：低报警信息说明；
  - Low Low：低低报警信息说明。

点击报警栏右侧的[...]，进入说明设定，如图 20-19 所示。

- Rate Of Change 关于正反变化率的信息说明：

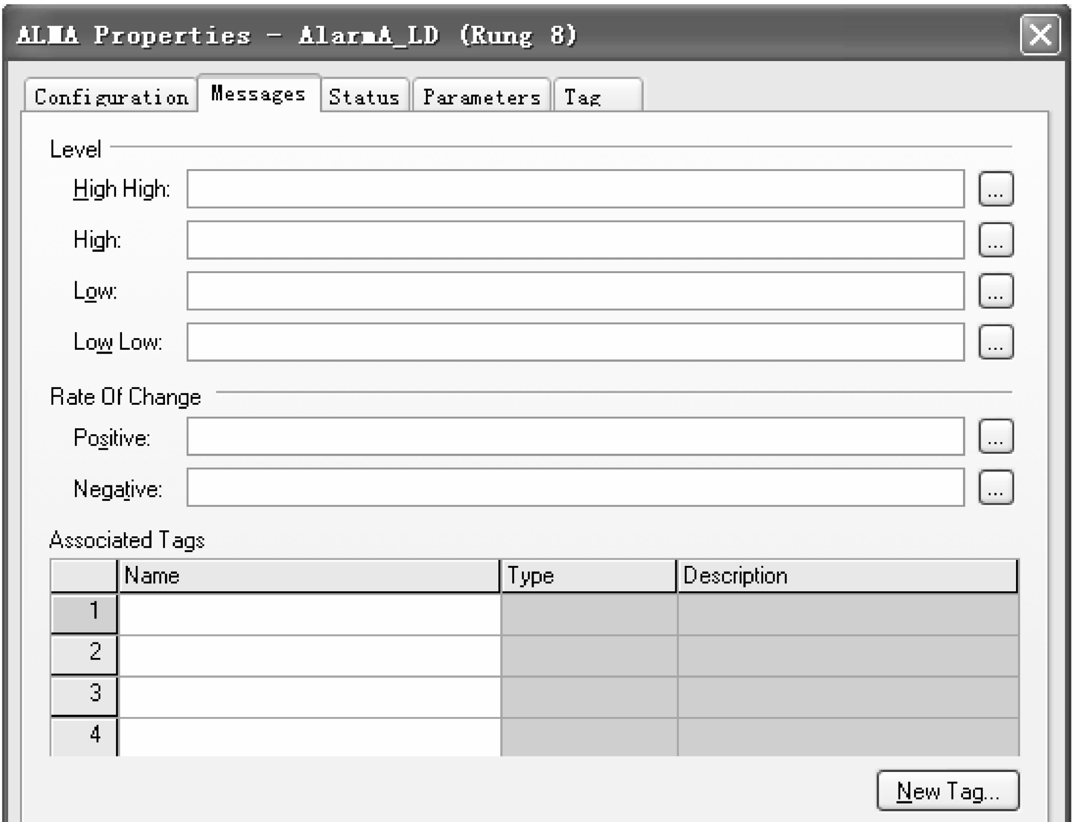


图 20-18　模拟量报警信息组态页面

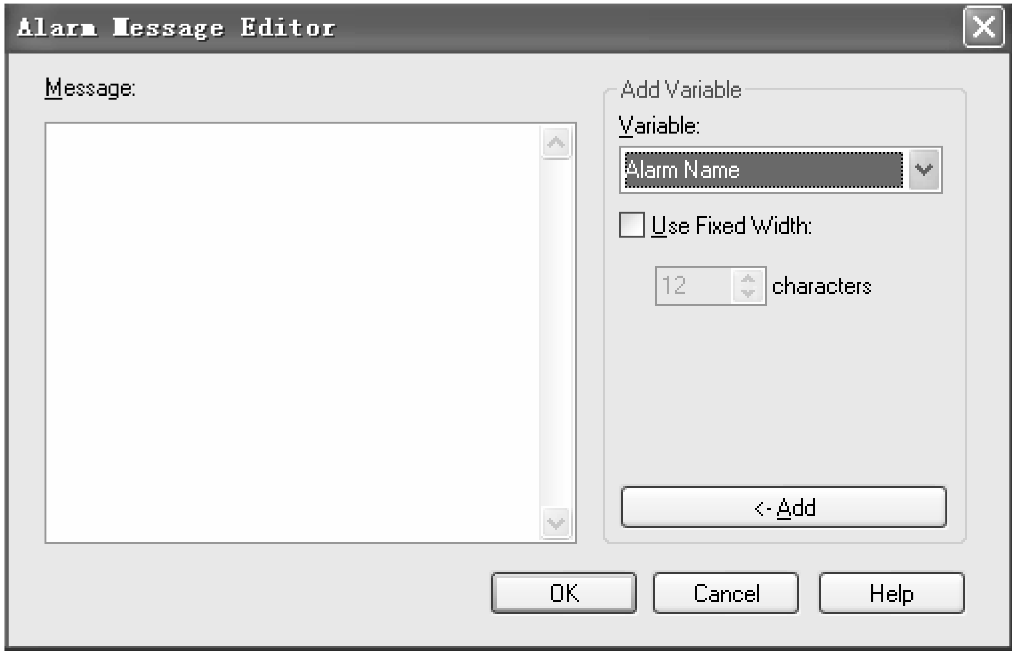


图 20-19　报警信息说明设定

- Positive：正向变化率报警信息说明；
- Negative：反向变化率信息说明。

● Associated Tag（关联标签）：指定最多 4 个与报警有关联的标签，任何时候这些标签被存储的值映射于此，报警条件变成报警状态时，或其他报警状态发生，这些存储值被传送给报警。这些存储的值也可以被内含在报警信息中。关联标签可以是基本数据或字符串，也可以是用户自定义标签的或数组的子元素。

点击 Status 页面，如图 20-20 所示是页面左侧的关于高高报警、高报警、低报警和低低报警的时间记录。如图 20-21 所示的是页面右侧的关于正向变化率报警和反向变化率报警的时间记录。

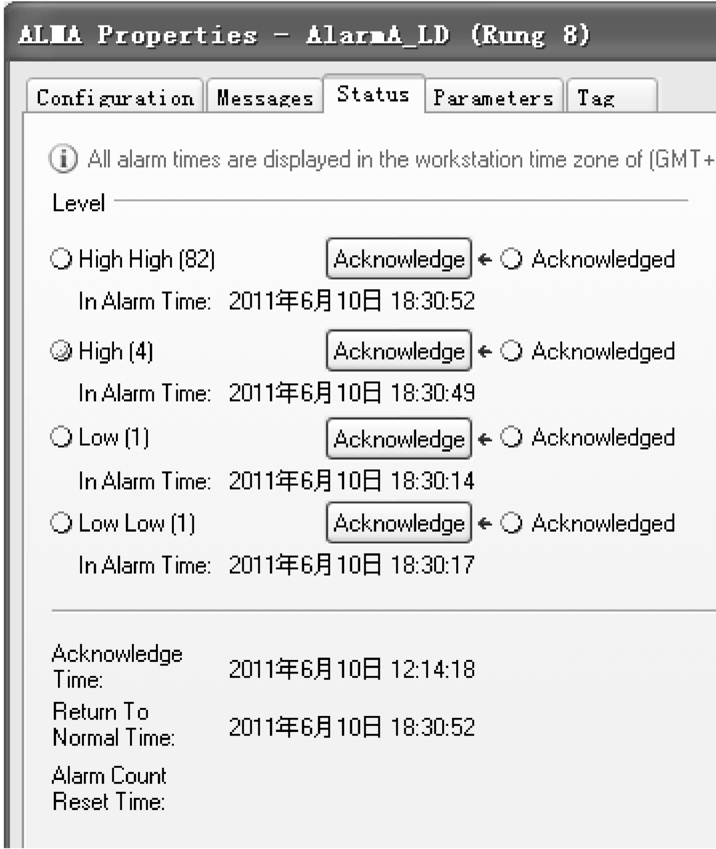


图 20-20　报警状态左侧的时间记录

如图 20-22 所示是参数页面的部分输入参数。此组态页面所见到的只是其中很少的一部分，关于每个参数的含义和运用，在罗克韦尔出版物的指令集中可以找到详细的介绍。

如图 20-23 所示是参数页面的部分输出参数。此组态页面所见到的只是其中很少的一部分，关于每个参数的含义和运用，在罗克韦尔出版物的指令集中可以找到详细的介绍。

运行测试，当模拟量输入数据 27648 超过高报警值 25000 时，延时 3s 后，相应的报警位 HInAlarm 置位，同时相应的确认位 HAcked 复位，其余另外 3 个确认位依然在置位状态，如图 20-24 所示。

模拟量报警的功能块组态如图 20-25 所示。

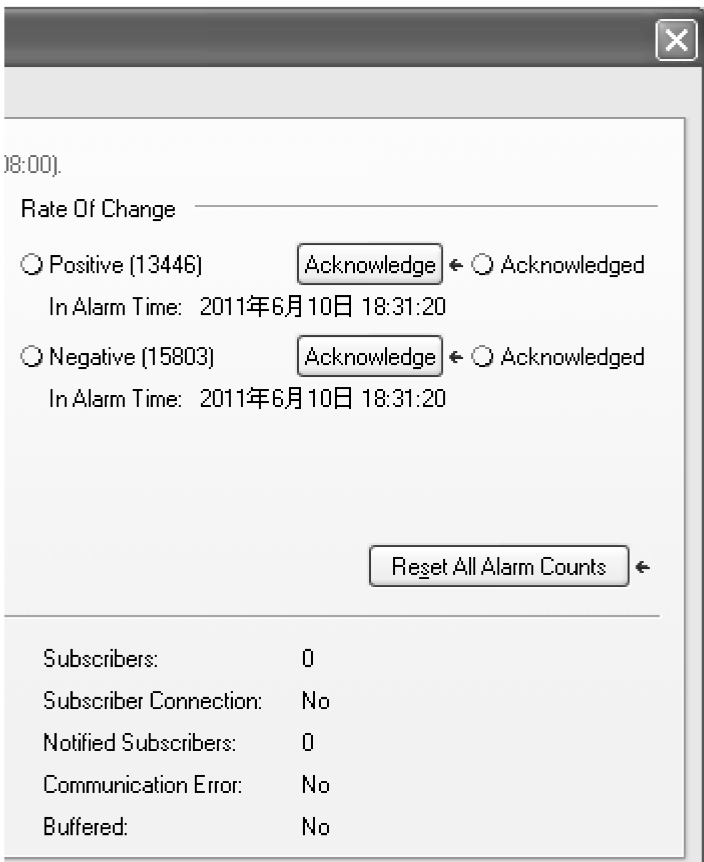
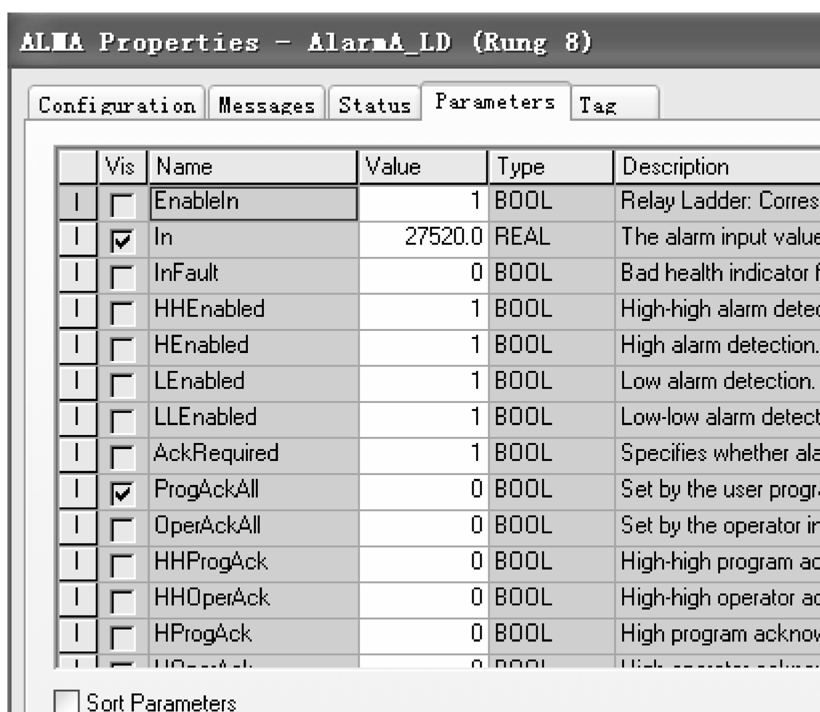


图 20-21　报警状态右侧的时间记录

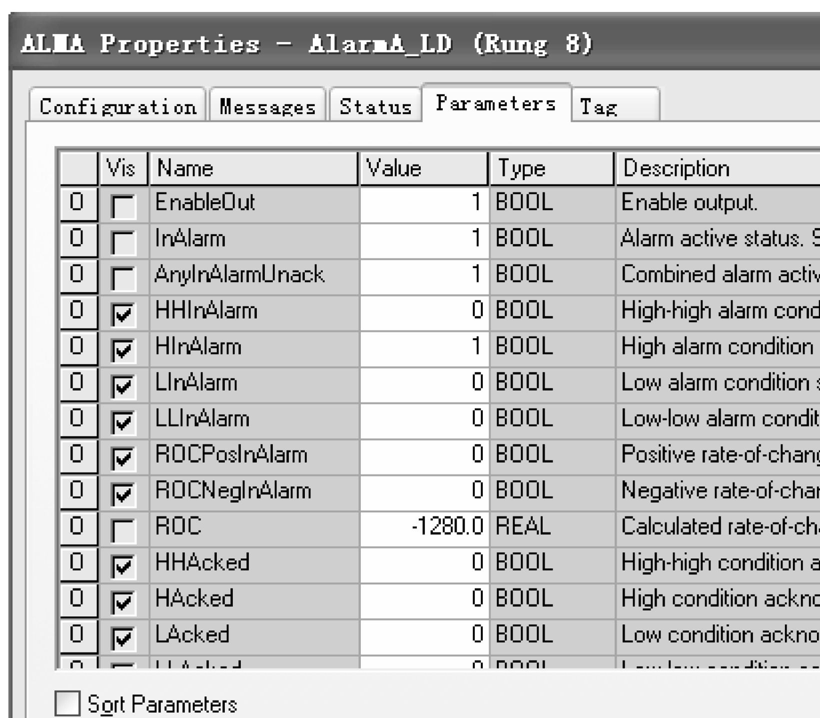
外部显示参数：

- 输入 In，模拟量报警监测对象，为模拟量模块 3 通道；
- 报警状态 HHInAlarm 为 1 表示处在高高报警中，为 0 表示未报警；
- 报警状态 HInAlarm 为 1 表示处在高报警中，为 0 表示未报警；
- 报警状态 LInAlarm 为 1 表示处在低报警中，为 0 表示未报警；
- 报警状态 LLInAlarm 为 1 表示处在低低报警中，为 0 表示未报警；
- 报警状态 ROCPosInAlarm 为 1 表示处在正向变化率报警中，为 0 表示未报警；
- 报警状态 ROCNegInAlarm 为 1 表示处在反向变化率报警中，为 0 表示未报警；
- 报警已确认 HHAcked 为 1 表示高高报警已被确认，为 0 表示未确认；
- 报警已确认 HAcked 为 1 表示高报警已被确认，为 0 表示未确认；
- 报警已确认 LAcked 为 1 表示低报警已被确认，为 0 表示未确认；
- 报警已确认 LLAcked 为 1 表示低低报警已被确认，为 0 表示未确认；
- 报警已确认 ROCPosAcked 为 1 表示正向变化率报警已被确认，为 0 表示未确认；
- 报警已确认 ROCNegAcked 为 1 表示反向变化率报警已被确认，为 0 表示未确认；
- 报警禁止 Suppressed 为 1 表示报警被禁止，为 0 表示未被禁止；
- 报警消除 Disabled 为 1 表示报警取消，为 0 表示报警使能；



	Vis	Name	Value	Type	Description
I	☐	EnableIn	1	BOOL	Relay Ladder: Corres
I	☑	In	27520.0	REAL	The alarm input value
I	☐	InFault	0	BOOL	Bad health indicator f
I	☐	HHEnabled	1	BOOL	High-high alarm detec
I	☐	HEnabled	1	BOOL	High alarm detection.
I	☐	LEnabled	1	BOOL	Low alarm detection.
I	☐	LLEnabled	1	BOOL	Low-low alarm detect
I	☐	AckRequired	1	BOOL	Specifies whether ala
I	☑	ProgAckAll	0	BOOL	Set by the user progr
I	☐	OperAckAll	0	BOOL	Set by the operator in
I	☐	HHProgAck	0	BOOL	High-high program ac
I	☐	HHOperAck	0	BOOL	High-high operator ac
I	☐	HProgAck	0	BOOL	High program acknov

图 20-22　参数页面的部分输入参数



	Vis	Name	Value	Type	Description
O	☐	EnableOut	1	BOOL	Enable output.
O	☐	InAlarm	1	BOOL	Alarm active status. S
O	☐	AnyInAlarmUnack	1	BOOL	Combined alarm activ
O	☑	HHInAlarm	0	BOOL	High-high alarm condi
O	☑	HInAlarm	1	BOOL	High alarm condition s
O	☑	LInAlarm	0	BOOL	Low alarm condition s
O	☑	LLInAlarm	0	BOOL	Low-low alarm conditi
O	☑	ROCPosInAlarm	0	BOOL	Positive rate-of-chang
O	☑	ROCNegInAlarm	0	BOOL	Negative rate-of-char
O	☐	ROC	-1280.0	REAL	Calculated rate-of-cha
O	☑	HHAcked	0	BOOL	High-high condition a
O	☑	HAcked	0	BOOL	High condition ackno
O	☑	LAcked	0	BOOL	Low condition acknov

图 20-23　参数页面的部分输出参数

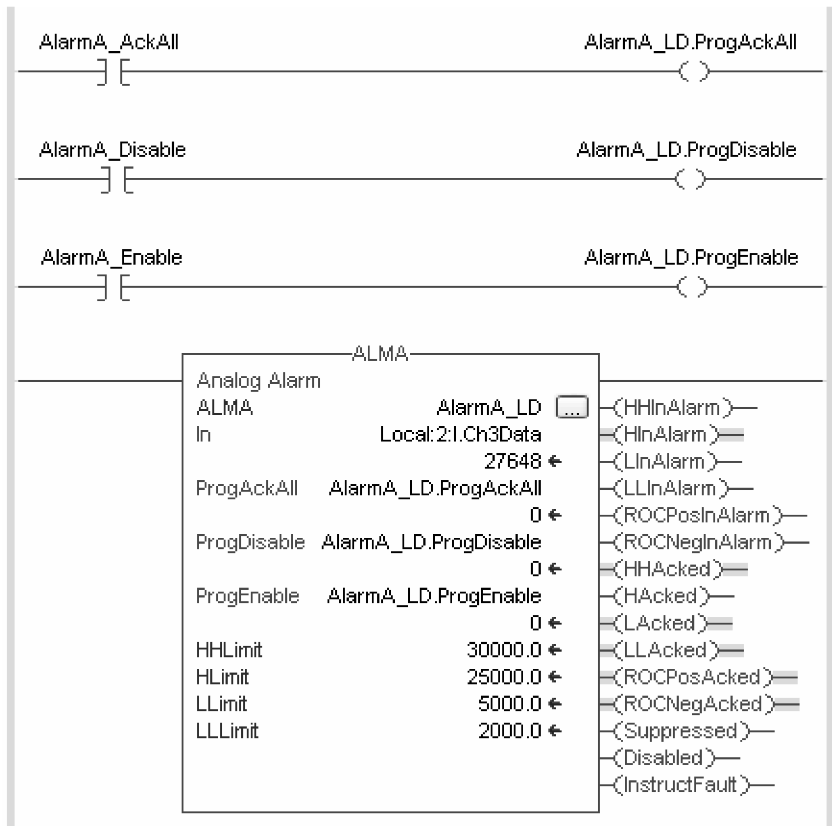


图 20-24　模拟量报警梯级逻辑

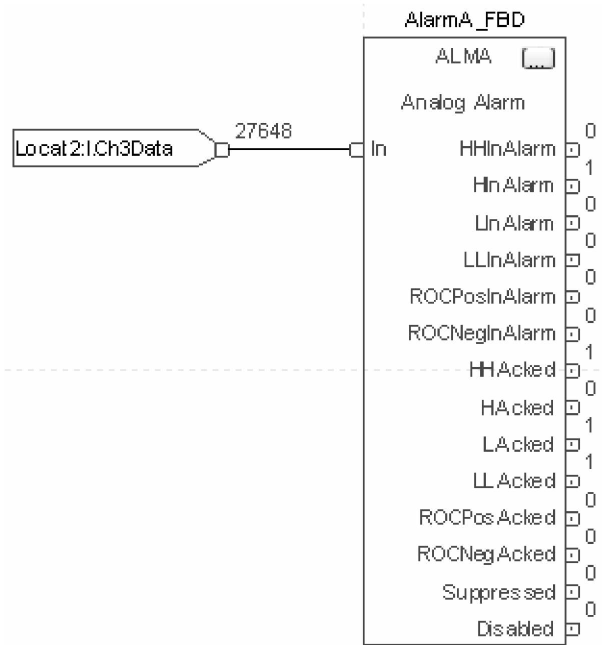


图 20-25　模拟量报警的功能块组态

● 报警指令故障 InstructFault 为 1 表示指令故障，为 0 表示指令正常。

点击[...]后，进入组态页面，跟梯形图指令完全一样，选择相同的设置，这里不再赘述。

运行测试，当模拟量输入数据 27392 超过高报警值 25000 时，延时 3s 后，相应的报警位 HInAlarm 置位，同时相应的确认位 HAcked 复位，其余另外 3 个确认位依然在置位状态，如图 20-26 所示。

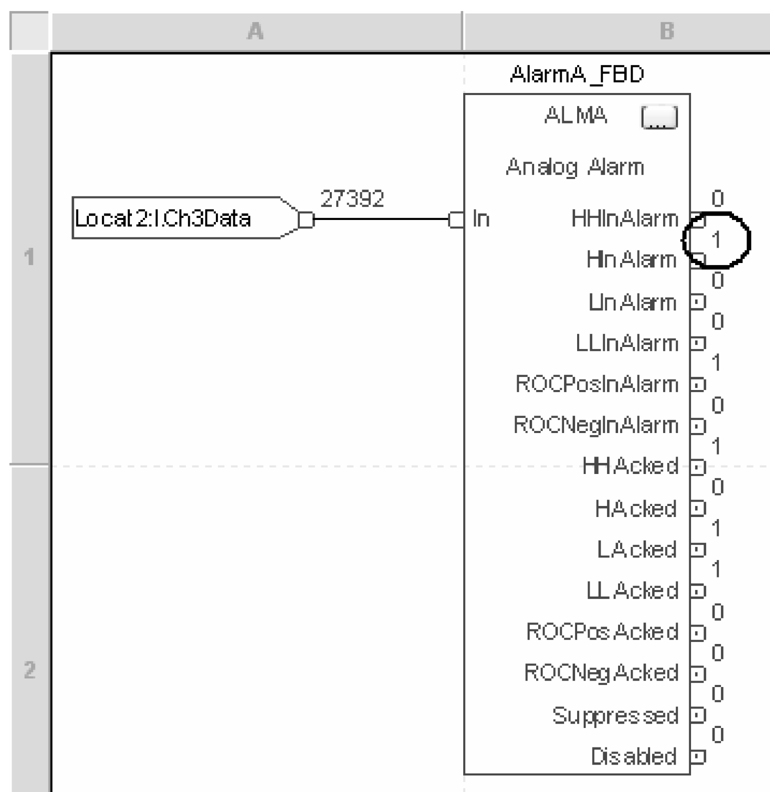


图 20-26　功能块 ALMA 执行的结果

最后，我们再来看看报警功能的语句编程。首先，在数据库创建一个模拟量报警的结构数据标签，命名为 AlarmA _ Text，如图 20-27 所示。

编写语句报警逻辑如图 20-28 所示。赋值语句完成了模拟量输入数据、模拟量报警的所有报警确认、报警使能和报警取消的程控操作，指令 ALMA 及圆括号中相关的参数顺序而列。

语句编程的报警页面组态不能在离线情况下完成，只有在线控制器中可直接进行组态，在语句指令圆括号的参数项中，选中报警结构标签 AlarmA _ Text，点击右键，选择 Configure “AlarmA _ Text”，点击进入，即可进入跟梯形图指令和功能块完全一样的组态界面，从而完成组态工作。语句编程的报警页面组态的进入如图 20-29 所示。

同样，报警的组态必须在模拟量报警结构数据标签 AlarmA _ Text 中完成，不管是梯形图指令编程、功能块编程，抑或是语句编程，模拟量报警结构数据都是一样的，不同的是梯形图逻辑和功能块有特定的界面，可直观地完成。模拟量结构数据如图 20-30 所示，如果熟

练的话，直接在结构数据表中设置有关参数也未尝不可。

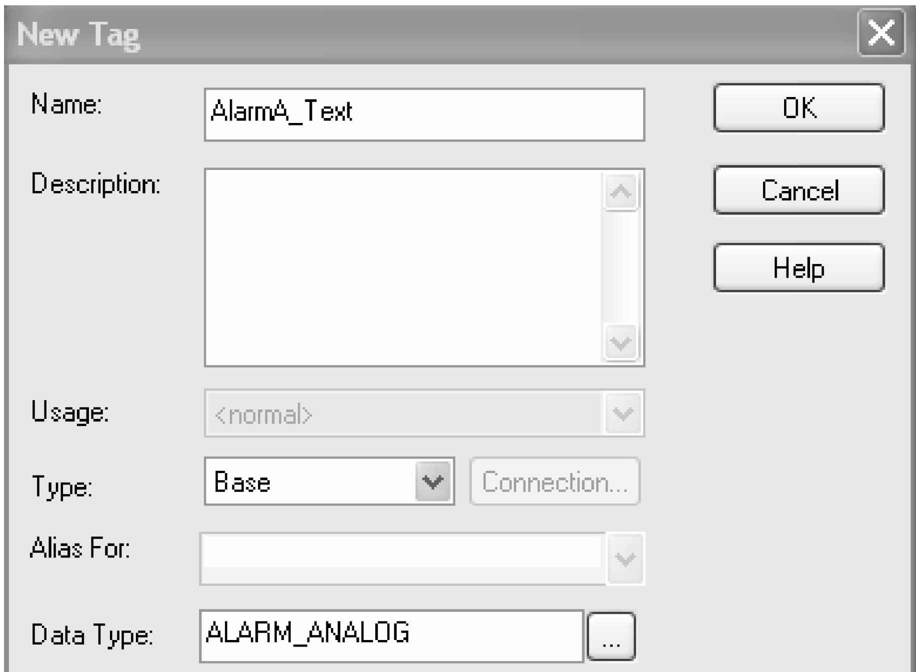


图 20-27　在数据库创建一个模拟量报警的结构数据标签

```
AlarmA_Text.In := Local:2:I.Ch3Data;
AlarmA_Text.ProgAckAll:= AlarmA_AckAll;
AlarmA_Text.ProgDisable := AlarmA_Disable;
AlarmA_Text.ProgEnable := AlarmA_Enable;
ALMA (AlarmA_Text,AlarmA_Text.In,AlarmA_Text.ProgAckAll,
AlarmA_Text.ProgDisable,AlarmA_Text.ProgEnable);
```

图 20-28　编写的语句模拟量报警

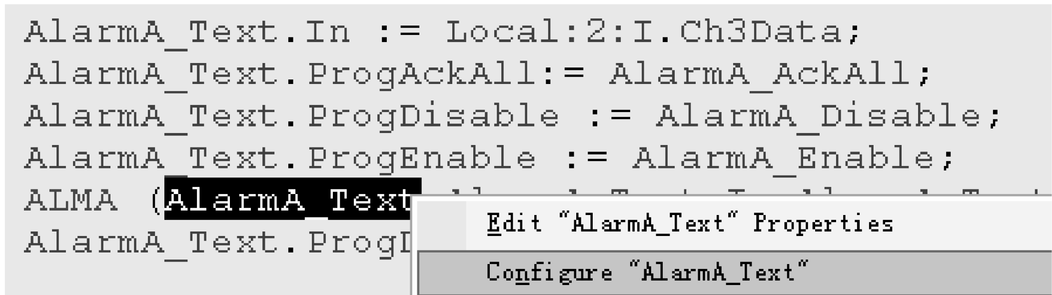


图 20-29　语句编程的报警页面组态的进入

⊟ AlarmA_Text	{...}	{		ALARM_ANA
AlarmA_Text.EnableIn	0		Decimal	BOOL
AlarmA_Text.In	0.0		Float	REAL
AlarmA_Text.InFault	0		Decimal	BOOL
AlarmA_Text.HHEnabled	1		Decimal	BOOL
AlarmA_Text.HEnabled	1		Decimal	BOOL
AlarmA_Text.LEnabled	1		Decimal	BOOL
AlarmA_Text.LLEnabled	1		Decimal	BOOL
AlarmA_Text.AckRequired	1		Decimal	BOOL
AlarmA_Text.ProgAckAll	0		Decimal	BOOL
AlarmA_Text.OperAckAll	0		Decimal	BOOL
AlarmA_Text.HHProgAck	0		Decimal	BOOL
AlarmA_Text.HHOperAck	0		Decimal	BOOL
AlarmA_Text.HProgAck	0		Decimal	BOOL
AlarmA_Text.HOperAck	0		Decimal	BOOL
AlarmA_Text.LProgAck	0		Decimal	BOOL
AlarmA_Text.LOperAck	0		Decimal	BOOL
AlarmA_Text.LLProgAck	0		Decimal	BOOL
AlarmA_Text.LLOperAck	0		Decimal	BOOL
AlarmA_Text.ROCPosProgAck	0		Decimal	BOOL
AlarmA_Text.ROCPosOperAck	0		Decimal	BOOL
AlarmA_Text.ROCNegProgAck	0		Decimal	BOOL
AlarmA_Text.ROCNegOperAck	0		Decimal	BOOL
AlarmA_Text.ProgSuppress	0		Decimal	BOOL
AlarmA_Text.OperSuppress	0		Decimal	BOOL
AlarmA_Text.ProgUnsuppress	0		Decimal	BOOL
AlarmA_Text.OperUnsuppress	0		Decimal	BOOL
AlarmA_Text.ProgDisable	0		Decimal	BOOL
AlarmA_Text.OperDisable	0		Decimal	BOOL
AlarmA_Text.ProgEnable	0		Decimal	BOOL
AlarmA_Text.OperEnable	0		Decimal	BOOL
AlarmA_Text.AlarmCountReset	0		Decimal	BOOL
AlarmA_Text.HHLimit	0.0		Float	REAL
⊞ AlarmA_Text.HHSeverity	500		Decimal	DINT
AlarmA_Text.HLimit	0.0		Float	REAL
⊞ AlarmA_Text.HSeverity	500		Decimal	DINT
AlarmA_Text.LLimit	0.0		Float	REAL
⊞ AlarmA_Text.LSeverity	650		Decimal	DINT
AlarmA_Text.LLLimit	0.0		Float	REAL
⊞ AlarmA_Text.LLSeverity	500		Decimal	DINT
⊞ AlarmA_Text.MinDurationPRE	0		Decimal	DINT
AlarmA_Text.Deadband	0.0		Float	REAL

图 20-30　模拟量报警结构数据标签

AlarmA_Text.ROCPosLimit	0.0	Float	REAL
AlarmA_Text.ROCPosSeverity	500	Decimal	DINT
AlarmA_Text.ROCNegLimit	0.0	Float	REAL
AlarmA_Text.ROCNegSeverity	500	Decimal	DINT
AlarmA_Text.ROCPeriod	0.0	Float	REAL
AlarmA_Text.EnableOut	0	Decimal	BOOL
AlarmA_Text.InAlarm	0	Decimal	BOOL
AlarmA_Text.AnyInAlarmUnack	0	Decimal	BOOL
AlarmA_Text.HHInAlarm	0	Decimal	BOOL
AlarmA_Text.HInAlarm	0	Decimal	BOOL
AlarmA_Text.LInAlarm	0	Decimal	BOOL
AlarmA_Text.LLInAlarm	0	Decimal	BOOL
AlarmA_Text.ROCPosInAlarm	0	Decimal	BOOL
AlarmA_Text.ROCNegInAlarm	0	Decimal	BOOL
AlarmA_Text.ROC	0.0	Float	REAL
AlarmA_Text.HHAcked	1	Decimal	BOOL
AlarmA_Text.HAcked	1	Decimal	BOOL
AlarmA_Text.LAcked	1	Decimal	BOOL
AlarmA_Text.LLAcked	1	Decimal	BOOL
AlarmA_Text.ROCPosAcked	1	Decimal	BOOL
AlarmA_Text.ROCNegAcked	1	Decimal	BOOL
AlarmA_Text.HHInAlarmUnack	0	Decimal	BOOL
AlarmA_Text.HInAlarmUnack	0	Decimal	BOOL
AlarmA_Text.LInAlarmUnack	0	Decimal	BOOL
AlarmA_Text.LLInAlarmUnack	0	Decimal	BOOL
AlarmA_Text.ROCPosInAlarmUnack	0	Decimal	BOOL
AlarmA_Text.ROCNegInAlarmUnack	0	Decimal	BOOL
AlarmA_Text.Suppressed	0	Decimal	BOOL
AlarmA_Text.Disabled	0	Decimal	BOOL
AlarmA_Text.MinDurationACC	0	Decimal	DINT
AlarmA_Text.HHInAlarmTime	DT#19...	Date/Ti...	LINT
AlarmA_Text.HHAlarmCount	0	Decimal	DINT
AlarmA_Text.HInAlarmTime	DT#19...	Date/Ti...	LINT
AlarmA_Text.HAlarmCount	0	Decimal	DINT
AlarmA_Text.LInAlarmTime	DT#19...	Date/Ti...	LINT
AlarmA_Text.LAlarmCount	0	Decimal	DINT
AlarmA_Text.LLInAlarmTime	DT#19...	Date/Ti...	LINT
AlarmA_Text.LLAlarmCount	0	Decimal	DINT
AlarmA_Text.ROCPosInAlarmTime	DT#19...	Date/Ti...	LINT
AlarmA_Text.ROCPosAlarmCount	0	Decimal	DINT
AlarmA_Text.ROCNegInAlarmTime	DT#19...	Date/Ti...	LINT
AlarmA_Text.ROCNegAlarmCount	0	Decimal	DINT

图 20-30　模拟量报警结构数据标签（续）

AlarmA_Text.AckTime	DT#19...		Date/Ti...	LINT
AlarmA_Text.RetToNormalTime	DT#19...		Date/Ti...	LINT
AlarmA_Text.AlarmCountResetTime	DT#19...		Date/Ti...	LINT
AlarmA_Text.DeliveryER	0		Decimal	BOOL
AlarmA_Text.DeliveryDN	0		Decimal	BOOL
AlarmA_Text.DeliveryEN	0		Decimal	BOOL
AlarmA_Text.NoSubscriber	0		Decimal	BOOL
AlarmA_Text.NoConnection	0		Decimal	BOOL
AlarmA_Text.CommError	0		Decimal	BOOL
AlarmA_Text.AlarmBuffered	0		Decimal	BOOL
+ AlarmA_Text.Subscribers	0		Decimal	DINT
+ AlarmA_Text.SubscNotified	0		Decimal	DINT
+ AlarmA_Text.Status	0		Decimal	DINT
AlarmA_Text.InstructFault	0		Decimal	BOOL
AlarmA_Text.InFaulted	0		Decimal	BOOL
AlarmA_Text.SeverityInv	0		Decimal	BOOL
AlarmA_Text.AlarmLimitsInv	0		Decimal	BOOL
AlarmA_Text.DeadbandInv	0		Decimal	BOOL
AlarmA_Text.ROCPosLimitInv	0		Decimal	BOOL
AlarmA_Text.ROCNegLimitInv	0		Decimal	BOOL
AlarmA_Text.ROCPeriodInv	0		Decimal	BOOL

图 20-30　模拟量报警结构数据标签（续）

报警功能的讨论中，大家可能已经注意到，我们很少谈论指令编程的技巧，更多的是在讨论参数的含义，这和我们最初讨论的内容是不一样的，这说明系统越成熟，指令功能越强，标准化处理越多，编程人员所需要的技巧越少，编程人员的负担越轻，人为出错的因素越少，项目开发的时间就越短。这不正是我们追求和期望的吗？

参 考 文 献

［1］ Logix 控制器指令集 . 罗克韦尔自动化公司资料 . 2010.

［2］ Logix 控制器用户手册 . 罗克韦尔自动化公司资料 . 2010.

电气自动化新技术丛书

序号	书名	书号	定价	出版时间
1	SPWM 变频调速应用技术（第 4 版）			
2	通用变频器及其应用（第 3 版）	35756-8		201109
3	交流永磁电机变频调速系统	33589-4	39.8	201105
4	变频调速 SVPMW 技术的原理、算法与应用	31903-0	38	201104
5	高性能变频调速及其典型控制系统	30268-1	49	201106
6	智能建筑设备自动化系统	29452-8	47	201003
7	电压型 PWM 整流器的非线性控制	25144-6	27	200901
8	高压大功率变频调速技术	19218-3	30	200710
9	现场总线技术及其应用（第 2 版）	14269-0	47	200808
10	谐波抑制和无功功率补偿（第 2 版）	06298-1	29	201101
11	电气传动的脉宽调制控制技术（第 2 版）	04453-3	22	200402
12	直流无刷电动机原理及应用（第 2 版）	05038-4	17	200710

变频器系列

序号	书名	书号	定价	出版时间
1	变频调速技术基础教程			
2	施耐德变频器的应用	35783-4		201110
3	施耐德变频器原理与应用	26954-0	47	201001
4	变频器应用教程（第 2 版）	34130-7	40	201106
5	生产机械的变频调速	33189-6	69.8	201106
6	变频器应用与维修	32496-6	45	201103
7	变频器应用图册	31948-1	29.8	201101
8	变频器原理与维修	29808-3	46	201101
9	变频器电路维修与故障实例分析	28319-5	38	201103
10	变频器实用电路图集与原理图说	26924-3	29	201008
11	西门子系列变频器及其工程应用	24328-1	36	201008
12	感应电动机传动和变频器应用技术	23059-5	20	200803
13	变频器应用手册（第 3 版）	21185-3	43	200802
14	常用变频器功能手册	14979-3	27	200603
15	调速用变频器及配套设备选用指南（第 2 版）	07808-X	40	200608
16	变频器、PLC 在仿制工业中的应用	26876-5	50	200908
17	变频器、PLC 及组态软件实用技术速成教程	29856-4	59.8	201101
18	变频器、可编程序控制器及触摸屏综合应用技术	18581-9	65	201009
19	S7-300 PLC 和 MM440 变频器的原理与应用	19667-9	33	201103

自动化软件

序号	书名	书号	定价	出版时间
1	自动测试系统测试描述语言	34114-7	38	201107
2	AutoCAD 电气设计快速入门与提高	32360-0	40	201101
3	从基础到实践——PLC 与组态王	34776-7	49.8	201108
4	西门子 S7 可编程序控制器——STEP7 编程指南（第 2 版）（1CD）	28718-6	66	201009
5	西门子 STEP7 编程语言与使用技巧（1CD）	26848-2	48	201001
6	STEP7 开发基础及应用指南	25158-3	38	200901
7	专业电气绘图软件 PCschematic ELautomation 中文教程	21177-8	49	201001

工业网络与现场总线

序号	书名	书号	定价	出版时间
1	PLC 网络系统配置指南	34132-1	69.8	201107
2	三菱电机通信网络应用指南	30673-3	59	201009
3	西门子 PLC 与工业网络技术（含 1CD）	23355-8	39	201101
4	西门子工业网络通信指南（上册）（1CD）	15177-7	49	201004
5	西门子工业网络通信指南（下册）（1CD）	16508-8	58	200908
6	西门子工业网络交换机应用指南	24211-6	53	200806
7	从 Modbus 到透明就绪——施耐德电气工业网络的协议、设计、安装和应用（1CD）	25490-4	68	200901
8	现场总线技术及应用教程——从 PROFIBUS、RROFINET 到 As-i	20529-6	38	201102
9	工业控制网络与现场总线技术	18421-8	28	201101

数控技术

序号	书名	书号	定价	出版时间
1	数控机床调试诊断与维修	35223-5	49.8	201108
2	数控机床故障诊断与维修	27930-3	34	201108
3	图解 FANUC PMC 编程与应用	32289-4	59	201101
4	大隈 OSP 数控系统的故障分析	29898-4	48	201007
5	数控机床电气维修笔记——积累、经验和心得	27512-1	35	200910
6	实用数控技术	26521-4	39.8	200910
7	数控机床编程与操作	26778-2	33	201104
8	现代数控系统——原理、构成与实例	18781-3	39	200702


机械工业出版社 CHINA MACHINE PRESS 电工电子分社

图书订购与咨询热线

图书零售：

网上订购：中国科技金书网
（机械工业出版社旗下大型科技图书网站）
网址：www.golden-book.com
E-mail：service@golden-book.com

电话订购：010-68993821　010-88379639/9641/9643
传真订购：010-68990188
邮局汇款：
地址：北京市 100037 信箱 30 分箱
收款人：网络销售部
邮编：100037
银行转帐：
户名：北京百万庄图书大厦有限公司
帐号：341556020680
开户行：中国银行北京百万庄支行

图书团购：

联系人：李红勇
电话：010-88379766
Email：dgdzcmp@sina.com

Rockwell Automation

罗克韦尔自动化技术丛书

书　名	书　号	定 价
1、循序渐进Power Flex变频器	20376-6	33.00
2、循序渐进Kinetix集成运动控制系统	24498-1	45.00
3、电气控制与可编程序控制器应用技术	22095-4	49.00
4、ControlLogix系统实用手册	23037-3	58.00
5、循序渐进SLC500控制系统与PanelView训练课	23038-0	45.00
6、深入浅出NetLinx网络架构	24983-2	30.00
7、ControlLogix系统电力行业自动化应用培训教程	24127-0	58.00
8、循序渐进CMS机器控制系统	25797-4	47.00
9、ControlLogix系统水泥行业自动化应用培训教程	26077-6	39.00
10、MicroLogix核心控制系统	31237-6	59.00
11、ControlLogix系统在污水处理行业中的应用	35489-5	45.00
12、ControlLogix系统在给水处理行业中的应用	35490-1	45.00
13、PAC编程基本教程	36030-8	85.00

ISBN 978-7-111-36030-8
策划编辑：林春泉／封面设计：鞠杨
定价：85.00元

地址：北京市百万庄大街22号
电话服务
社服务中心：(010)88361066
销售一部：(010)68326294
销售二部：(010)88379649
读者购书热线：(010)88379203
邮政编码：100037
网络服务
门户网：http://www.cmpbook.com
教材网：http://www.cmpedu.com
封面无防伪标均为盗版

上架指导：电气工程／自动化技术

